A MEMORY-AWARE SCHEDULING FRAMEWORK FOR STREAMING APPLICATIONS ON MULTICORE SYSTEMS

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2019

By
Mingze Ma
School of Computer Science

Contents

Abstract 10

Declaration 11

Copyright 12

The Author 13

Acknowledgements 14

Notations 15

1 Introduction 17

1.1 Scheduling for Multicore Systems 17
1.2 Streaming Applications 18
1.3 Synchronous Dataflow Graphs 20
1.4 Problem Statement 21
1.5 A Memory-Aware Scheduling Framework 22
1.6 Contributions 24
1.7 Thesis Overview 25

2 Background and Literature Review 26

2.1 Overview 26
2.2 Synchronous Dataflow Graphs 26
2.2.1 The Concept of an SDFG 26
2.2.2 Scheduling for SDFGs 28
2.3 Basic Properties of SDFGs 29
2.3.1 Connectedness 29
2.3.2 Topology Matrix 30

2.3.3 Consistency and Liveness 31
2.3.4 SDFG-to-HSDFG Conversion 34
2.3.5 Iteration Period and Throughput 34
2.4 SDFG Transformation Techniques 35
2.4.1 Retiming 35
2.4.2 Unfolding 38
2.4.3 Combinational Use of Retiming and Unfolding 43
2.5 Problem Models 44
2.5.1 Ideal Model 45
2.5.2 The Number of Processing Cores 45
2.5.3 Communication Architecture 46
2.5.4 Heterogeneity of Processors 46
2.5.5 Scheduling Objectives 47
2.6 Analysis and Modelling Approaches 49
2.6.1 Modelling Buffer Bound 50
2.6.2 Modelling Auto-Concurrencies 51
2.6.3 Modelling Scheduling and Mapping 51
2.6.4 Modelling Communication Overhead 53
2.7 Scheduling Methodologies 54
2.7.1 Blocked Scheduling 54
2.7.2 Solver Based Periodic Scheduling 55
2.7.3 Evolutionary Algorithm based Scheduling 57
2.7.4 Self-Timed Execution Based Scheduling 57
2.8 Scheduling Algorithms 63
2.8.1 Scheduling Algorithms for Model $MD(X|F,N_s)$ 63
2.8.2 Scheduling Algorithms for Model $MD(X|F)$ 67
2.8.3 Scheduling Algorithms for Model $MD(X|F,N)$ 70
2.8.4 Scheduling Algorithms for Model $MD(X|F,N,C)$ 75
2.9 Evaluation Tools and Benchmarks for SDFGs 81
2.10 Summary 82

3 Buffer Minimization for Rate-Optimal Scheduling 83

3.1 Overview 83
3.2 Background 85
3.2.1 Problem Definition 85
3.2.2 Buffer-Constrained Self-Timed Execution 86

3.3 Motivational Example 88
3.4 Algorithms 91
3.4.1 Extended Buffer-Constrained Self-Timed Execution 92
3.4.2 Obtaining the Initial Point 93
3.4.3 Exact Algorithm for Buffer Size Minimization 94
3.4.4 A Heuristic for Buffer Size Minimization 96
3.5 Evaluation 97
3.5.1 Preliminaries 97
3.5.2 Results 98
3.6 Summary 100

4 Communication-Aware Scheduling for SDFGs 102
4.1 Overview 102
4.2 Background 105
4.2.1 Hardware Architecture 107
4.2.2 Objective and Criteria 108
4.3 Motivational Example 109
4.4 Algorithms 110
4.4.1 Communication-Aware Self-Timed Execution 110
4.4.2 Scheduling Algorithms 116
4.4.3 Extension for Heterogeneous Systems 118
4.5 Evaluation 119
4.5.1 Settings for Homogeneous Algorithms 119
4.5.2 Results for Homogeneous Algorithms 121
4.5.3 Settings and Results for Heterogeneous Algorithms 124
4.6 Summary 131

5 Code-Size-Aware Mapping for SDFGs 132
5.1 Overview 132
5.2 Problem Definition 134
5.3 Motivational Example 134
5.4 Algorithms 136
5.4.1 ILP Based Mapping Approach 136
5.4.2 Code-Size Aware Mapping Heuristic 139
5.4.3 Extension for Communication-Aware Scheduling 142
5.5 Evaluation 144

5.5.1 Experimental Setup 144
5.5.2 Realistic Applications 145
5.5.3 Randomly Generated SDFGs 147
5.5.4 The Trade-Off between Throughput and Code Size for Communication-Aware Scheduling 149
5.6 A Case Study 153
5.6.1 Setting 153
5.6.2 Scheduling 154
5.7 Summary 156

6 Conclusion and Future Work **157**
6.1 Conclusion 157
6.2 Future Work 159

List of Tables

2.1 Scheduling algorithms for the model $MD(X|F,N_s)$. 67
2.2 Scheduling algorithms for the model $MD(X|F)$. 69
2.3 Scheduling algorithms for the model $MD(X|F,N)$. 74
2.4 Scheduling algorithms for the model $MD(X|F,N,C)$. 80

3.1 The graph generation parameters. 97
3.2 Result improvement and runtime increase for randomly generated SDFGs. 98
3.3 Result improvement and runtime increase for the proposed heuristic algorithm. 99
3.4 Search Range Reduction. 100

4.1 Basic notations. 106
4.2 The number of actors N, the sum of the minimum repetition number of actors SR and the runtime of the SDFG-to-HSDFG conversion for each of the 12 realistic SDFGs used in the experiments 120
4.3 Speedup and runtime for all homogeneous algorithms and applications using 2, 4, 8 and 16 homogeneous cores. 122
4.4 Speedup and runtime for all heterogeneous algorithms and applications using heterogeneous cores with random speed. 125
4.5 Speedup and runtime for all heterogeneous algorithms and applications using low speed heterogeneous cores. 126
4.6 Speedup and runtime for all heterogeneous algorithms and applications using high speed heterogeneous cores. 127

5.1 Code size and IP got by three mapping methods. 135
5.2 Variables for the ILP Model. 136
5.3 The code size obtained by DES, CSAS and CSMS and the runtime of CSAS and CSMS for 8 realistic applications when $M = 2$. 145
5.4 The code size obtained by CSAS and DES for random SDFGs. 148

5.5 The code size reduction obtained by CCSAS for eight applications when throughput degradation is 95%, 90% and 80%. 153
5.6 The buffer usage obtained by NAIVE, ZHU and EXACT for BitonicSortRecursive. 154
5.7 The code size and speedup obtained by CCSAS using different values for the code size factor F_{cs}. 155

List of Figures

1.1 Developing trends of multiprocessors. [Rup18] 18
1.2 The SDFG of an H.263 encoder. [Stu07] 19
1.3 A memory-aware scheduling framework. 23

2.1 Properties about various scheduling strategies. [SB09] 28
2.2 An SDFG-to-HSDFG conversion example. 34
2.3 A retiming example. 36
2.4 The classic retiming and the general retiming of an example HSDFG. 37
2.5 An unfolding example. 38
2.6 Two counterexamples. 40
2.7 An examples to illustrate that unfolding an HSDFG by its optimal unfolding factor may lead to sub-optimal throughput under the constraint of a fixed number of processing cores. 42
2.8 A counterexample for using retiming alone. 43
2.9 A counterexample for using unfolding alone. 44
2.10 A counterexample for buffer bound modelling in [WBJS06] and [SGB06a]. 50
2.11 An example for modelling auto-concurrences using self-loop channels. 51
2.12 An example for modelling scheduling and mapping for an HSDFG. . 52
2.13 An example for modelling communication delays. 53
2.14 HSDFG-to-DAG conversion for an HSDFG. 54
2.15 An example SDFG and its STE. 61

3.1 An example SDFG. 84
3.2 The BC-STE of the example SDFG in Figure 3.1. 87
3.3 An example illustrating the limitation of [ZGBS14]. 89
3.4 The eBC-STE of the example SDFG in Figure 3.1. 92

4.1 An example SDFG. 102

4.2 The bus-based multicore architecture. 107
4.3 The transformation based scheduling. 108
4.4 The scheduling process of ERAS for the example SDFG in Figure 4.1. 108
4.5 The average speedup improvement of all algorithms compared with DLS using 2, 4, 8 and 16 cores for the 8 realistic applications which can be completed by all homogeneous algorithms. 121
4.6 The average speedup improvement of all algorithms compared with DLS using 2, 4, 8 and 16 cores for the 8 realistic applications which can be completed by all heterogeneous algorithms. 128

5.1 An example SDFG. 133
5.2 Three schedules of the example SDFG based on three mapping strategies respectively. 135
5.3 Code size reduction of CSAS for realistic SDFGs on multicore processors with 2, 4, 8 and 16 cores compared with DES. 146
5.4 The trade-off between the throughput and code size obtained by CCSAS for eight realistic applications. 152

Abstract

A MEMORY-AWARE SCHEDULING FRAMEWORK FOR STREAMING APPLICATIONS ON MULTICORE SYSTEMS
Mingze Ma
A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2019

As a result of the rapidly increasing requirements for computing capability, multicore systems are more and more widely used. However, the performance of multicore systems does not always increase linearly with the number of processing cores. Many practical factors are limiting the speed of a multicore system, e.g. inter-core communication overhead and finite parallelism within applications. The scheduling of an application decides the parallel execution of the application on a multicore system, which significantly determines the performance of the application.

Many multimedia applications and digital signal processing applications share the same property that these applications need to be executed iteratively to process a continuous input data stream. To describe the unique property of these applications, a dataflow model named synchronous dataflow graph (SDFG) is used. Different from commonly used models such as the well-known directed acyclic graph (DAG), the SDFG model is able to describe the inter-iteration dependencies and the multi-rate nature of streaming applications. Accordingly, SDFGs require dedicated scheduling approaches other than the well-developed scheduling approaches for DAGs.

In this thesis, a memory-aware synchronous dataflow graph scheduling framework is presented, where two significant criteria, data processing speed and the size of memory usage, of streaming applications are optimized. Effective algorithms used in the three key steps of the framework are given, including buffer minimization algorithms, communication-aware scheduling algorithms, and code-size-aware mapping algorithms. Experimental results show that the proposed algorithms outperform existing algorithms in terms of throughput and memory usage in most cases.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see `http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487`), in any relevant Thesis restriction declarations deposited in the University Library, The University Library's regulations (see `http://www.manchester.ac.uk/library/aboutus/regulations`) and in The University's policy on presentation of Theses

The Author

Mingze Ma was born in Heilongjiang, China on August 7, 1989. He received his bachelor's degree in IC Design and Intergraded Systems from Heilongjiang University, China in 2011. After that, he obtained the master's degree in IC engineering from Fudan University, China, in 2013 with an Outstanding Graduate Award (top 5% of all the graduates). The topic of his master's thesis is about task scheduling on parallel systems.

Since September of 2014, Mingze has started working toward the Ph.D. degree in computer science at the University of Manchester. During his Ph.D. period, he has published and presented four papers in peer-reviewed conferences. His research interests include compile-time task scheduling, multicore systems, streaming applications and synchronous dataflow models.

Acknowledgements

The production of this thesis should be thankful to many people, who I want to mention gratefully in this section.

Firstly, I would like to thank my supervisor Dr. Rizos Sakellariou, who has always been kind, supportive and respectable throughout the four years of my PhD study. I have learned a lot from Rizos about the techniques and moralities of becoming a qualified researcher through his words and deeds. He always encourages me to publish my work and to communicate with the academic communities on high-quality conferences. I have also gained valuable experience of academic paper publication while preparing papers with him. In addition, I should thank him for the precious comments and suggestions for the draft of this thesis.

Secondly, I am very thankful for all the members who worked or still are working in our office room 2.124, for example Abdelkhalik Mosa, Chong Ke and many other members. Thanks for creating such a harmonious atmosphere in the office and always being helpful whenever I came across any problem on life or research. I should also thank my schoolmates Qinhao Wu, Chen Qian and many others who would like to share their leisure time with me. I will miss the laughter and happiness at our gatherings.

Lastly, I would like to thank my parents in particular and all the relatives in my big family, alongside all my friends, who have always been my strongest backing in every way. Without their cultivation and support, I could not easily make any achievement in my life and become the person who I am today, let alone complete this thesis.

Notations

G	An SDFG
V	The set of all the actors in an SDFG
E	The set of all the channels in an SDFG
α	An actor in set V
$IN(\alpha)$	The set of input channels of the actor α
$OUT(\alpha)$	The set of output channels of the actor α
$ET(\alpha)$	The execution time of the actor α
$CS(\alpha)$	The code size of the actor α
$\Upsilon(\alpha)$	The minimum repetition number of the actor α
e	A channel in set E
$\alpha_{src}(e)$	The source actor of the channel e
$\alpha_{dst}(e)$	The destination actor of the channel e
$it(e)$	The number of initial tokens on the channel e
$\boldsymbol{IT}$	The vector of the number of initial tokens on channels
$TS(e)$	The data size of one unit token on the channel e
$p(e)$	The token producing rate of the source actor of the channel e
$q(e)$	The token consuming rate of the destination actor of the channel e
LB	The lower bound of the iteration period of an SDFG
P	A path in a directed graph
L_P	The length of a path
N	The number of actors in an SDFG
L	The number of channels in an SDFG
d	A buffer distribution, where an element $d(e)$ corresponds to the buffer size on the channel e
$RT(G)$	A retiming scheme of an SDFG G
$S(G)$	A periodic schedule of an SDFG G
$IP(S)$	The period of a periodic schedule $S(G)$

$J(S)$	The unfolding factor of a schedule $S(G)$
$TH(S)$	The throughput of a schedule $S(G)$
SR	The sum of the minimum repetition number of all actors
C	The set of all the processing cores in a multicore system
c	A processing core in C
$\boldsymbol{B}(e)$	The FIFO buffer on channel e
b	A token block in the FIFO buffer $\boldsymbol{B}(e)$, $b = (num, et, cid)$
$b(num)$	The number of tokens in the token block b
$b(et)$	The existing time of the token block b
$b(cid)$	The processing core which has produced the token block b
M	The number of processing cores
BW	The bandwidth of the bus

Chapter 1

Introduction

1.1 Scheduling for Multicore Systems

With the development of semiconductor technology in the past sixty years, the number of transistors on a chip doubled every two years as predicted by the famous Moore's Law [Sch97]. Figure 1.1 from [Rup18] shows the exponential increase in the number of transistors on one processor over the last 42 years, as well as the developing trends of processors in terms of single thread performance, frequency, power and the number of processing cores. As shown in Figure 1.1, the frequency of processors has not improved within the last decade because of the limitations in energy consumption and the requirement of heat dissipation. The performance of single processing cores for a single thread is restricted by the frequency of processors and increased relatively little in the last decade. As the performance requirements of applications are growing rapidly, multicore architectures are more and more widely used. According to Figure 1.1, since 2005, the number of processing cores in a processor increased exponentially from one to as many as hundreds nowadays. The development of the multicore architecture benefits from its high energy-efficiency. For example, techniques like Dynamic Power Management (DPM) [BBDM00] and Dynamic Voltage and Frequency Scaling (DVFS) [SMB^{+}02] enable the dynamic adjustment of energy consumption on different cores. Furthermore, the use of heterogeneous processing cores helps achieving both high performance and low power consumption with multicore architectures.

However, not every application can be executed in parallel. Even for applications suitable for parallel execution, the execution of these applications on multicore systems is decided by the adopted scheduling strategies, which finally determine the performance of these applications. Therefore, to make full use of the computing power of multicore

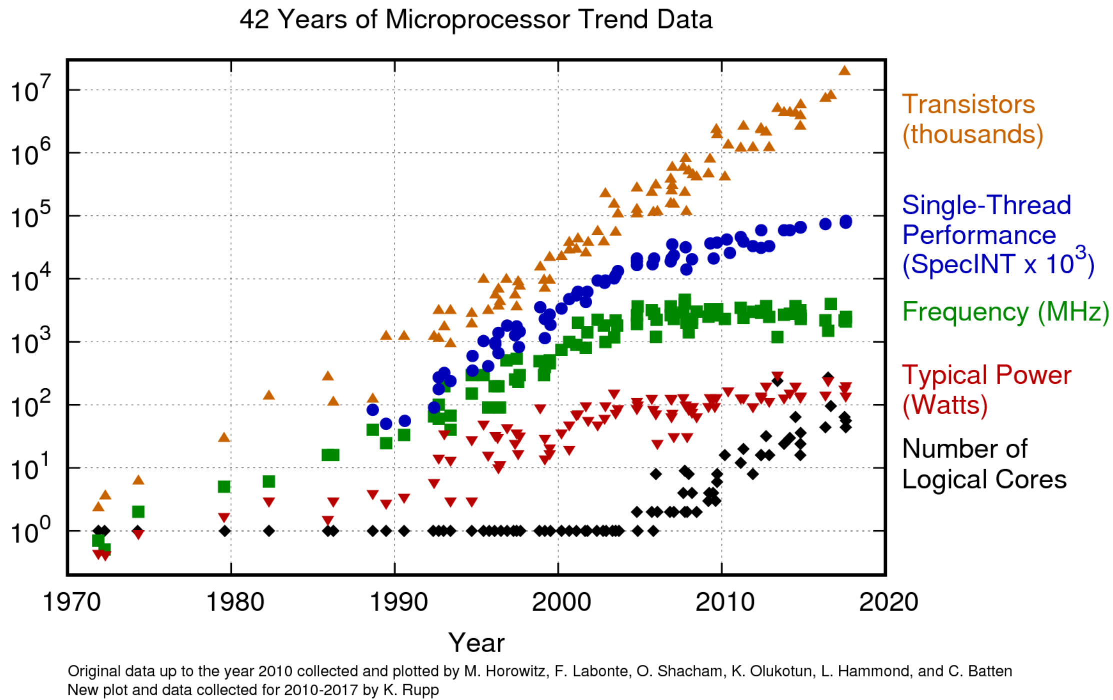


Figure 1.1: Developing trends of multiprocessors. [Rup18]

systems and the parallelism within an application, effective scheduling methodologies are desirable to be proposed. In this thesis, a scheduling framework is proposed for the scheduling of streaming applications on multicore systems. The performance and memory usage of applications are both optimized within the proposed scheduling framework.

1.2 Streaming Applications

In embedded systems, many digital signal processing applications and multimedia applications are typically used for processing continuous data streams, e.g., a video decoder keeps decoding the data stream of a video to get continuous output frames. These applications are called **streaming applications** [TKA02]. The target streaming applications in this thesis have some properties:

- Streaming applications always run iteratively, therefore the performance of these applications is often decided by the average period of many iterations rather than the finishing time of one single iteration;

- Streaming applications are module based. Data producing and consuming rates

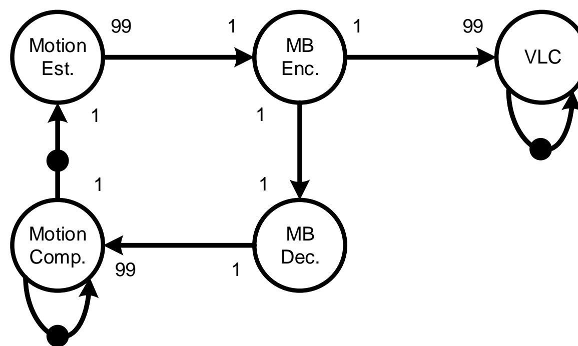


Figure 1.2: The SDFG of an H.263 encoder. [Stu07]

of modules are specified in advance and are not always the same with each other;

- Data dependencies between modules may exist within one iteration or across iterations;

- The computation of the application is relatively simple and the input data is massive.

These unique properties show that the behaviour of the target streaming applications may be predictable and suitable for compile-time scheduling.

There are three levels of parallelism existing in streaming applications, including **task level parallelism**, **data level parallelism** and **pipeline level parallelism**. The task level parallelism in streaming applications stems from the possibilities of executing different modules in parallel. For the data level parallelism, as the data producing and consuming rates of modules in a streaming application can be different, a module may be invoked multiple times within one iteration to process a series of different input data. Therefore, the multiple invocations of the same module enable the existence of the data level parallelism in a streaming application. Furthermore, since streaming applications are executed iteratively, the possibility of executing different iterations of a streaming application in parallel results in the pipeline level parallelism. These three levels of parallelism make streaming applications to be highly suitable for parallel execution on multicore systems.

1.3 Synchronous Dataflow Graphs

To describe the unique properties of streaming applications, a well-known dataflow model, called Synchronous Dataflow Graph (SDFG), is proposed in [LM+87a] and widely used for the scheduling of streaming applications. An H.263 encoder from [Stu07] is shown in Figure 1.2 as an example SDFG. The nodes of the SDFG are called **actors**; they represent the functional modules of an application. The directed edges between actors are called **channels**; they represent data dependencies between actors. Communication data is modelled as **tokens** with a different token size. The token producing and consuming rates of actors are labeled next to the two ends of each channel. The channel with **initial tokens** (i.e., the black dots on channels) represents an inter-iteration data dependency. The initial tokens also enable the existence of cycles in SDFGs and prevent deadlocks. To guarantee the correctness of the execution of an SDFG, the destination actor of the channel has to consume the earliest produced tokens first. Therefore, token transfer on a channel is modelled using a first-in-first-out (FIFO) **buffer** on that channel.

For the H.263 encoder shown in Figure 1.2, the actors Motion Est., Motion Comp., MB Enc., MB Dec. and VLC represent the motion estimator module, the motion compensation module, the macro block (MB) encoding module, the MB decoding module and the variable length encoder module of the H.263 encoder, respectively. Here, the MB encoding module consists of a discrete cosine transformation (DCT) and a quantizer; and the MB decoding module is implemented with an inverse DCT (IDCT). The H.263 encoder divides a video frame into 99 micro blocks. Actors Motion Est., Motion Comp. and VLC process a video frame in one invocation, while actors MB Enc. and MB Dec. only process one micro block in one invocation. In Figure 1.2, one micro block is modelled as one token, and the difference of the data processing rates of actors is represented by the variety of the token producing and consuming rates on channels. The edges leading to themselves for the actors Motion Comp. and VLC imply that there are global variables or internal static variables in these actors which are updated in each invocation of these actors. Therefore, the execution of these actors is dependent on their previous invocations.

1.4 Problem Statement

Although streaming applications always possess abundant parallelism and are suitable for parallel execution on multicore systems, the execution performance of these applications is still significantly dependent on the quality of the scheduling [SB09]. Since streaming applications process input data streams iteratively and continuously, the criterion for the performance of a streaming application is not the finishing (execution) time of one iteration of the application, but **throughput**, which represents the data processing rate of the application. Throughput optimization is one of the most significant objectives of SDFG scheduling on multicore systems.

Although much work has been proposed to optimize the throughput of schedules for SDFGs on multicore systems, all existing work has limitations.

- Reference [LM87b] suggests the conversion of SDFGs to directed acyclic graphs (DAGs) and the use of scheduling algorithms for DAGs to schedule the obtained DAGs. However, the number of actors and edges of an SDFG can be increased exponentially after the SDFG has been converted to a DAG. In addition, the inter-iteration parallelism of SDFGs cannot be utilized directly by scheduling algorithms for DAGs.

- References [Ram74, BMKdD12, MBGS10, ZGBS14] propose scheduling algorithms for SDFGs without converting SDFGs to DAGs. However, the constraint of the number of processing cores is not considered in these papers.

- Some work develops approaches to schedule SDFGs under the constraint of the number of processing cores, yet the inter-core communication overhead is not considered, e.g. references [DGGH92, BLMB09, BLMB13, FKBS11, ZGBS12, TBG$^+$17a, LKOH13].

- For communication-aware scheduling, most existing work only tries to solve a simplified scheduling problem for SDFG. For example, references [LGY15, KBB06, RS17] only aim at single-rate SDFGs and references [COKH12, LGE12, SBGC07, ZSJ10, DKV12, DKV14, DKV16, TBG$^+$17b] have the restriction that one actor can only be mapped on one core.

- The work [MG13] solves the communication-aware scheduling problem without simplification using a model-checking based approach. However, this approach is extremely time-consuming and not scalable for large-size SDFGs.

Above all, a fast and effective scheduling approach for SDFGs is desirable on multicore systems when the communication overhead is not negligible.

Except for the computing speed of processing cores, another factor restricting the performance of a computing system is the well-known memory wall [WM95], which is caused by the mismatching between the fast processing speed of the CPUs and the slow accessing speed of memory. To alleviate the bottleneck, fast on-chip memory such as hierarchical caches or scratchpad memory [BSL$^+$02] is widely used in processors. Compared with off-chip memory accessing, on-chip memory accessing is exponentially faster and more energy-saving. Therefore, the performance of a computing system can be improved by reducing the accessing of off-chip memory. On one hand, the size of on-chip memory is usually limited by the expensive on-chip space, therefore increasing the size of on-chip memory is not always an affordable choice. On the other hand, by reducing the requirements of memory usage of applications, accessing off-chip memory can still be effectively reduced or even avoided. Thus, reduction in memory use is also a widely studied problem for SDFG scheduling on multicore systems. For single core systems, the memory occupied by the buffers on channels and the code of actors is optimized in [BL93, TZB98, MB01, KMB07, CC13, ODH06, SKH98, Bjo04]. For multicore systems, there is a trade-off between the throughput and buffer size for SDFG scheduling. References [MBGS10, SGB06a, ZGBS12, ZGBS14, SBGC07, CZ12, SOH11, RS14] consider both buffer size and throughput during scheduling. However, no existing work has proposed any solution to get the minimum buffer size for SDFGs under the constraint of the optimal throughput. Furthermore, on multicore systems, the code size of an SDFG could be increased by the actor duplication on various cores, as an actor may be executed on multiple cores. Therefore, memory reduction approaches for both buffer size and code size are desirable for SDFGs on multicore systems.

In summary, the problem to be solved in this thesis is how to optimize the throughput and reduce the memory usage for SDFGs on multicore systems considering communication overhead.

1.5 A Memory-Aware Scheduling Framework

To solve the target problem of this thesis, a memory-aware scheduling framework is proposed to improve the throughput and memory usage for SDFGs on multicore systems. The target hardware platform of this framework is a bus-based multicore processor,

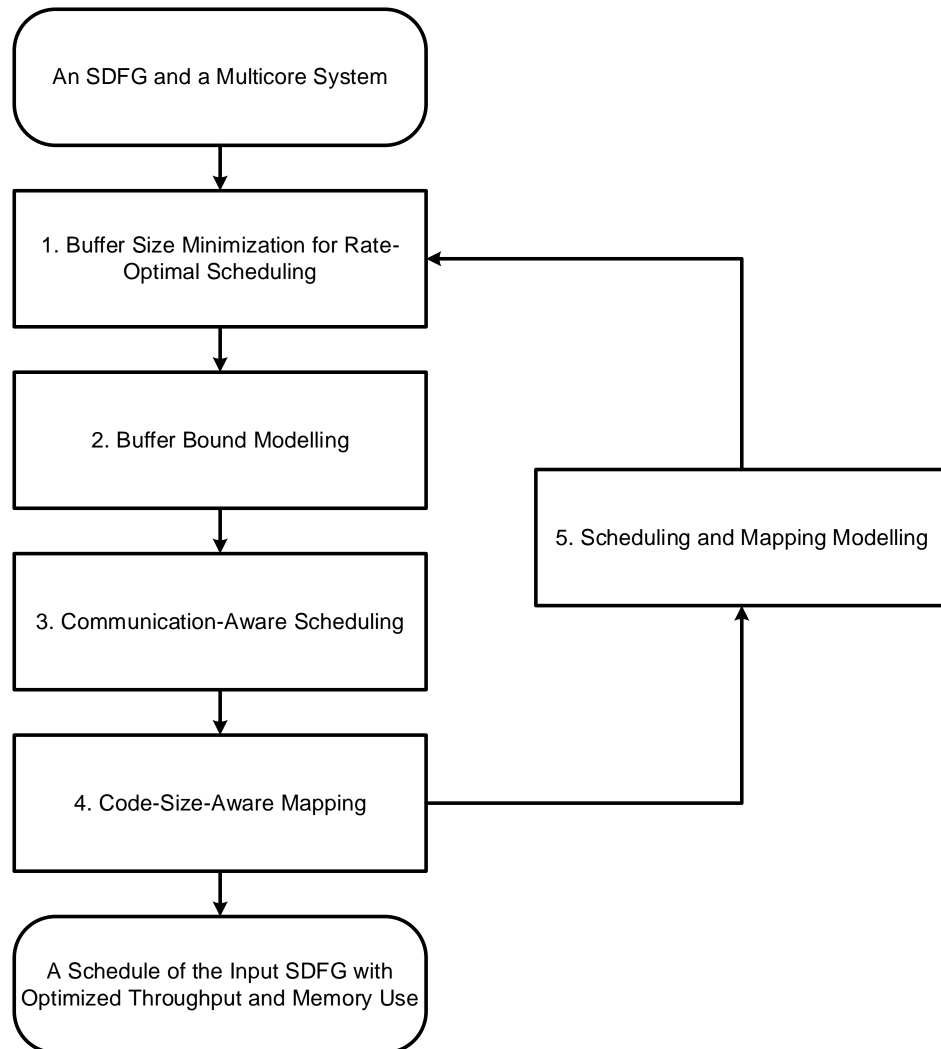


Figure 1.3: A memory-aware scheduling framework.

where each processing core has a private memory for both code and data buffering. Data communication between processing cores and off-chip memory accessing of processing cores are both through an arbitrated bus.

The steps of the proposed framework are shown in Figure 1.3. **Steps 1**, **3** and **4** are the **key steps** to be solved in this thesis (in Chapter 3, 4 and 5, respectively), while **Steps 2** and **5** can be implemented using existing modelling techniques introduced in references [ZGBS14], [BKKB02], [SBGC07], etc.

As shown in Figure 1.3, in **Step 1**, the proposed framework minimizes the buffer size under the constraint of the maximum throughput of the input SDFG. The number of processing cores is not considered in Step 1, which means an SDFG is assumed to be scheduled on infinite cores. The obtained buffer distribution is modelled as a buffer constraint in **Step 2**. After that, in **Step 3**, a communication-aware scheduling algorithm is used to get a schedule for the buffer-constrained SDFG. Then, in **Step 4**, a code-size-aware mapping process is conducted to optimize the code size of the obtained schedule. If the obtained schedule does not meet the design requirement, an iterative process could be performed to refine the obtained schedule. The obtained schedule can be modelled in **Step 5**. Then, the buffer size of the obtained schedule can be further reduced in **Step 1**. Furthermore, in **Step 3**, four (for homogeneous multicore systems) or two (for heterogeneous systems) interchangeable scheduling algorithms are proposed as alternatives if one scheduling algorithm cannot get any schedule that meets the design requirement. In addition, in **Step 4**, if the communication overhead is considered, there is a trade-off between the throughput and code size obtained by the proposed mapping process. Significant code size reduction can usually be achieved with minor throughput sacrifice.

1.6 Contributions

The contributions of this thesis are listed below.

- For **Step 1** of the proposed framework, an extended self-timed execution method is proposed to eliminate the extra buffer requirements caused by initial tokens. Based on this method, an exact algorithm and a heuristic are proposed to find a rate-optimal schedule with minimal buffer requirements for an SDFG based on the searching algorithm in [SGB06a]. This work was published in [MS16] and introduced in Chapter 3.

- For **Step 3** of the proposed framework, an extension of self-timed execution is proposed to support communication-aware scheduling of SDFGs, based on which four communication-aware scheduling algorithms are proposed for homogeneous multicore systems. This work was published in [MS18a]. The proposed algorithms for homogeneous systems are further extended to support heterogeneous systems in Chapter 4.

- For **Step 4** of the proposed framework, code-size-aware mapping approaches are proposed to reduce the code size of the scheduling of SDFGs on multicore systems. The early phase of this work was published in [MS17, MS18b]. The full version of this work is presented in Chapter 5, including an integer linear programming based exact solution and a code-size-aware mapping heuristic, and an extension of the proposed heuristic to support communication-aware scheduling.

1.7 Thesis Overview

The remainder of this thesis is organised as follows.

- Chapter 2 introduces the concepts used in this thesis, the properties of SDFGs, the scheduling models of SDFGs, the scheduling methodologies of SDFGs, and some useful SDFG transformation, analysis and modelling techniques. A detailed literature review is also given in Chapter 2.

- After that, the proposed buffer minimization approach is illustrated in Chapter 3.

- In Chapter 4, four communication-aware scheduling algorithms are proposed for SDFGs on homogeneous multicore systems. Two of these algorithms are extended to support heterogeneous multicore systems.

- In Chapter 5, code-size-aware mapping approaches are presented to reduce the extra code size brought by the actor duplication on multicore systems. Furthermore, a case study is presented as an example of applying the proposed framework.

- Lastly, Chapter 6 concludes and gives some future directions for this thesis.

Chapter 2

Background and Literature Review

2.1 Overview

This chapter introduces concepts and properties of Synchronous Dataflow Graphs (SDFGs) and defines the problem of scheduling SDFGs on multicore systems. SDFG transformation techniques and SDFG modelling techniques usually adopted as assistive approaches for SDFG scheduling, are also introduced. Lastly, related work for SDFG scheduling is given to show the state-of-the-art progress and open problems to be solved.

The concept of an SDFG and the principles of scheduling for SDFGs are presented in Section 2.2. The properties of SDFGs are introduced in Section 2.3. Two SDFG transformation techniques, retiming and unfolding, are introduced in Section 2.4. The problem models of SDFG scheduling and the objectives of the scheduling are given in Section 2.5. Section 2.6 introduces some modelling approaches of memory bounds and scheduling decisions. Section 2.7 summarizes scheduling methodologies for SDFGs, followed by a review for scheduling algorithms in Section 2.8. The evaluation tools and benchmarks used in this thesis are finally presented in Section 2.9.

2.2 Synchronous Dataflow Graphs

2.2.1 The Concept of an SDFG

An SDFG consists of **actors** and **channels**. Actors represent the computing task modules in an application. The data dependencies between actors are modelled by channels connecting them. Communication data is modelled as **tokens** with various

token size on channels. The channel with **initial tokens** (or **delays**) represents an inter-iteration data dependency. The initial tokens must be properly set to prevent deadlocks for cycles in SDFGs.

A graph theory based definition of an SDFG is given in Definition 1, followed by the definition of homogeneous SDFG (HSDFG), a special case of SDFG.

Definition 1 (SDFG)**.** An SDFG G is a tuple (V,E), where V is a finite set of actors and E is a finite set of directed channels connecting the actors in set V. An actor $\alpha \in V$ is a tuple (IN, OUT), where IN is the set of input channels for α and OUT is the set of output channels. A channel $e \in E$ is a tuple $(\alpha_{src}, \alpha_{dst}, p, q, it)$, where α_{src} and α_{dst} are the two actors connected to the channel while actor α_{dst} depends on actor α_{src} (i.e. $\alpha_{src} \rightarrow \alpha_{dst}$), p is the token producing rate of actor α_{src} on channel e, q is the token consuming rate of actor α_{dst} on channel e, and it is the number of initial tokens on the channel.

Definition 2 (HSDFG)**.** For an SDFG $G = (V,E)$, if $\forall e \in E$, we get $p(e) = q(e) = 1$, then this SDFG is referred to as a homogeneous SDFG.

For a channel e, we call the actor α_{src} as the **source actor** for the channel and the actor α_{dst} as the **destination actor** actor for the channel. α_{src} is a **predecessor** of actor α_{dst}, and actor α_{dst} is a **successor** of actor α_{src}. As the token producing rate of the source actor and the token consuming rate of the destination actor for a channel may not be the same, the tokens produced by the source actor may not be consumed immediately by the destination actor, which means the tokens produced by multiple executions of the source actors may accumulate on the channel. To guarantee the correctness of the execution of an SDFG, the destination actor of the channel has to consume the earliest produced tokens first. Therefore, the token transferring on a channel is modelled to be through a first-in-first-out (FIFO) buffer on the channel.

The definition of SDFG given in Definition 1 only describes the topology structure of an SDFG. However, to evaluate the performance of an application, the execution time of actors and the token size on channels should also be included in a detailed description of the application. This detailed description of an SDFG is denoted by a **timed SDFG**.

Definition 3 (Timed SDFG)**.** A timed SDFG G^T is an extension of an SDFG $G = (V,E)$. An extra element ET is added to the tuple of each actor in V, where ET is the execution time of an actor. Another element TS is added to the tuple of each channel in E, where TS is the size of a token on a channel.

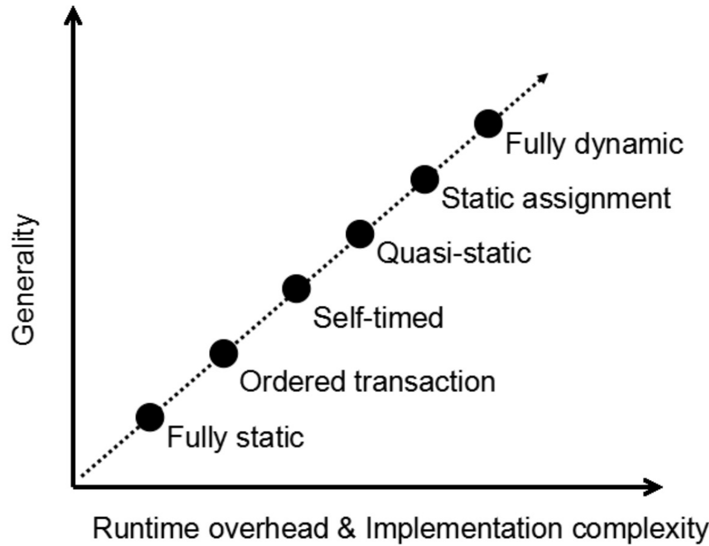


Figure 2.1: Properties about various scheduling strategies. [SB09]

2.2.2 Scheduling for SDFGs

Scheduling an SDFG on a multicore system includes handling the core assignment of actors, the execution of the actors on each processor, and the inter-processor communications. The scheduling can be categorized by how much scheduling work is done by static compilers and runtime schedulers. **Fully static** (or compile-time) scheduling means all scheduling work is done by compilers. In contrast, **fully dynamic** scheduling completely relies on runtime schedulers. Figure 2.1 from [SB09] shows some properties about various scheduling strategies, where the generality represents the range of applications that could be well handled by a scheduling strategy.

The **self-timed** scheduling only decides the assignment and the execution order of actors on each processor during compiling and does not care about the exact starting time of actors. For self-timed scheduling, the inaccurate estimation of the computing and communication time of actors does not affect the correctness of the output data, but may decrease the performance of the processing. **Ordered-transactions** scheduling basically does the same work as self-timed scheduling, except that the determination of the order for the inter-processor communications should also be made at compile time. **Quasi-static** scheduling can support input-dependent applications which may behave differently for different input. By statically profiling the execution of an input-dependent application under different input data, different static scheduling decisions can be made by a compiler to deal with each execution situation. **Static assignment** scheduling only

does the actor assignment at compile time and determine the execution of the actors by runtime schedulers.

Since the behaviour of SDFGs is predictable, static scheduling is used as the scheduling strategy in this thesis. In addition, static scheduling is also preferable for real-time embedded systems as it can provide guaranteed performance, which is always crucial for real-time systems. Besides, static schedules can be easily converted to or become a part of other scheduling strategies other than fully dynamic scheduling. For example, the starting time of actors of a static schedule can be used to generate the order of actors for self-timed scheduling.

Another dimension of the categorization for scheduling could be preemptive scheduling or non-preemptive scheduling. For **non-preemptive scheduling**, the execution of an actor cannot be interrupted if its execution has started. In contrast, **preemptive scheduling** enables the execution of an actor to be interrupted by the execution of another actor requiring the same computing resources. In this thesis, only non-preemptive scheduling is considered since the overhead brought by the interruption and recovery of the execution of actors in preemptive scheduling could be significant.

2.3 Basic Properties of SDFGs

2.3.1 Connectedness

Firstly, the concept of a path in a directed graph is given.

Definition 4 (Path)**.** A path $P = (e_1, e_2, ..., e_n)$ in a directed graph (V, E) is a finite, nonempty sequence of directed edges in E, where $\alpha_{dst}(e_i) = \alpha_{src}(e_{i+1}), (i = 1, 2, ..., n-1)$ and $\forall i \neq j, e_i \neq e_j, (i, j = 1, 2, ..., n)$.

For a path $P = (e_1, e_2, ..., e_n)$, the **length of a path** is defined as $L_P = ET(\alpha_{src}(e_1)) + \sum_{e \in P} ET(\alpha_{dst}(e))$. Then, two basic concepts in graph theory, **connected graph** and **strongly connected graph**, are given.

Definition 5 (Connected Graph)**.** If a graph only consists of one actor, the graph is connected. If a directed graph has two or more actors, and for any actor pair in the graph, there is a path that connects the two actors in the actor pair, then the graph is connected.

Definition 6 (Strongly Connected Graph)**.** If a directed graph has two or more actors, and every actor in the directed graph can get to any other actor in the graph through a

path, then the graph is defined as a strongly connected graph. A special case is that if a graph only consists of one actor, the graph is strongly connected.

A disconnected graph can be divided into strongly connected subgraphs, which can be viewed as independent graphs and analyzed separately. Here, the definition of the **subgraph** of a directed graph is given.

Definition 7 (Subgraph)**.** If a graph $G' = (V', E')$ is a subgraph of a directed graph $G = (V, E)$, then $V' \subseteq V$, and for all $e \in E$, $\alpha_{src}(e) \in V' \wedge \alpha_{dst}(e) \in V' \iff e \in E'$.

According to the definitions of the strongly connected graph and the subgraph, any directed graph can be separated into a series of strongly connected subgraphs with or without interconnections of directed edges. A **strongly connected component** (SCC) G'_S of a directed graph G is a strongly connected subgraph such that no other strongly connected subgraph of the graph is able to contain G'_S.

For the verification of the connectedness of a graph, many algorithms are applicable like Kosaraju's algorithm [Sha81], Tarjan's strongly connected components algorithm [Tar72] and the path-based strong component algorithm [Dij76].

2.3.2 Topology Matrix

A topology matrix is proposed by Lee in [LM+87a] to describe the topology structure of an SDFG except for the information of initial tokens. The matrix is denoted by $\mathbf{\Gamma}$. The entry $\mathbf{\Gamma}(i, j)$ in the matrix is related to the token producing/consuming rate of actor j on channel i as shown in Equation 2.1. If the actor j is the source of the channel i, then $\mathbf{\Gamma}(i, j)$ is the producing rate of the actor j on channel i. Conversely, if the actor j is the destination of the channel i, then $\mathbf{\Gamma}(i, j)$ is the negative consuming rate of the actor j on channel i. If actor j does not connect with channel i, then $\mathbf{\Gamma}(i, j) = 0$. A special case is that an actor may have a **self-loop** channel, which means the source actor and the destination actor of the channel are the same. In this case, the corresponding element in the matrix should be $\mathbf{\Gamma}(i, j) = p(i) - q(i)$.

$$\mathbf{\Gamma}(i, j) = \begin{cases} p(i) - q(i), & if\ j = \alpha_{src}(i) = \alpha_{dst}(i) \\ p(i), & if\ j = \alpha_{src}(i) \neq \alpha_{dst}(i) \\ -q(i), & if\ j = \alpha_{dst}(i) \neq \alpha_{src}(i) \\ 0, & otherwise \end{cases} \tag{2.1}$$

2.3.3 Consistency and Liveness

Because of the existence of cycles and actors with various data producing/consuming rates in SDFGs, to ensure an SDFG having at least one feasible schedule, the **sample rate consistency** and **liveness** of the graph have to be verified before scheduling. The definitions of sample rate consistency and liveness of an SDFG are given below.

Definition 8 (Sample Rate Consistency)**.** An SDFG is sample rate consistent if and only if the execution of the SDFG will not cause tokens accumulating on buffers.

Definition 9 (Liveness)**.** An SDFG is live if and only if the iterative execution of the graph does not stop for the insufficiency of the input tokens for any actor. Otherwise, it is deadlocked.

A **legal periodic schedule** of an SDFG must be sample rate consistent and deadlock-free. A necessary condition for the existence of a legal periodic schedule for a connected SDFG is given in Theorem 1 [LM87b].

Theorem 1 ([LM87b])**.** A connected SDFG G has a legal periodic schedule only if $rank(\mathbf{\Gamma}) = N - 1$, where $\mathbf{\Gamma}$ is the topology matrix of the graph and N is the number of actors in G.

Based on Theorem 1, another theorem is proven in [LM87b] which is provided as Theorem 2.

Theorem 2 ([LM87b])**.** If the topology matrix $\mathbf{\Gamma}$ of a connected SDFG satisfies the condition $rank(\Gamma) = N - 1$, where N is the number of actors, there must be a positive integer vector $\mathbf{\Upsilon} \neq \mathbf{0}$ such that $\mathbf{\Gamma} \times \mathbf{\Upsilon} = \mathbf{0}$ where $\mathbf{0}$ is the zero vector.

The vector $\mathbf{\Upsilon}$ in Theorem 2 has a practical meaning. The smallest positive solution of $\mathbf{\Upsilon}$ is referred to as a **repetition vector** which is defined in Definition 10.

Definition 10 (Repetition Vector)**.** The repetition vector $\mathbf{\Upsilon}$ for an SDFG has N entries, where N is the number of actors and each element of $\mathbf{\Upsilon}$ specifies the minimum times of the invocations of an actor in one period of a periodic schedule.

According to the definition of the repetition vector, $\mathbf{\Upsilon}$ can be obtained by solving a system of linear equations $\mathbf{\Gamma} \times \mathbf{\Upsilon} = \mathbf{0}$. The smallest positive solution is the repetition vector $\mathbf{\Upsilon}$. However, considering that at most two non-zero items may exist in each row of the topology matrix $\mathbf{\Gamma}$, some simpler ways of getting the repetition vector are

possible to be developed rather than solving the system of linear equations in a generic way.

A method is specified by Lee in [LM87b] to obtain the repetition vector. Similar methods of getting the repetition vector are also adopted in [Stu07, ZGBS14, GGS+06]. If an SDFG $G = (V,E)$ has at least one legal periodic schedule, then every self-loop channel $e \in E$ should have the same token producing rate $p(e)$ and token consuming rate $q(e)$. Otherwise, if $p(e) > q(e)$, the tokens will accumulate on the channel e, while if $p(e) < q(e)$, the execution of the actor connecting to the self-loop channel e will be blocked by lack of tokens on channel e. Based on this, each row in the topology matrix for an SDFG can only have zero or two non-zero elements if the SDFG has at least one legal periodic schedule. For each row having two non-zero elements, the values of the two elements are the producing rate of the source actor of a channel and negative consuming rate of the destination actor of the channel, respectively. For this case, one equation in the equation system $\mathbf{\Gamma} \times \mathbf{\Upsilon} = \mathbf{0}$ can be represented as

$$p(e)\mathbf{\Upsilon}(\alpha_{dst}(e)) = q(e)\mathbf{\Upsilon}(\alpha_{src}(e)).$$

This equation is called as **balance equation** in some references [ZGBS16, ZGBS12, ZGBS14]. The repetition vector $\mathbf{\Upsilon}$ can be obtained by solving the balance equations for all channels. The method in [LM87b] sets an arbitrary element $\mathbf{\Upsilon}(\alpha)$ in $\mathbf{\Upsilon}$ to 1, and get the values of the other elements in $\mathbf{\Upsilon}$ one by one according to the balance equations. During the calculation of $\mathbf{\Upsilon}$, if the obtained element is not an integer, store the value of the element as the simplest fraction. This fraction can be obtained by dividing both numerator and denominator by their greatest common divisor. If $\mathbf{\Upsilon}(\alpha) \neq 1$ is deduced by the calculation process, then the graph is either not sample rate consistent or not deadlock-free. After all elements in $\mathbf{\Upsilon}$ have been obtained, the smallest positive integer solution of the equation system can be achieved by multiplying all elements in $\mathbf{\Upsilon}$ by the least common multiple of the denominators of these elements.

A sufficient condition for the existence of a legal periodic schedule for a connected SDFG is also given in [LM87b], where a concept of class S algorithm is used. The definition of the class S algorithm is given in Definition 11, followed by the sufficient condition in Theorem 3.

Definition 11 (Class S Algorithm[LM87b])**.** Given a positive integer vector $\mathbf{\Upsilon}$ that satisfies $\mathbf{\Gamma} \times \mathbf{\Upsilon} = \mathbf{0}$ and an initial state for the tokens on channels, an actor α is runnable at a given time if it has not been run $\mathbf{\Upsilon}(\alpha)$ times and running it will not cause the

Algorithm 1 Deadlock Checking [LM87b]

Input:
An SDFG $G = (V, E)$; the repetition vector $\mathbf{\Upsilon}$ of the graph;
Output:
If the scheduling is not deadlocked, return $true$, else return $false$;
Iteration:

```
1:  Declare a set B, where each element B(e) represent the number of tokens on the
    buffer of a channel e; the initial value of B(e) is set to it(e);
2:  while ∃ϒ(α) > 0, ϒ(α) ∈ ϒ do
3:    runable := false;
4:    for all ϒ(α) > 0 do
5:      if ∃B(e) − q(e) < 0, e ∈ INPUT(α) then
6:        continue;
7:      end if
8:      for all e ∈ INPUT(α) do
9:        B(e) := B(e) − q(e);
10:     end for
11:     for all e ∈ OUTPUT(α) do
12:       B(e) := B(e) + p(e);
13:     end for
14:     runable := true;
15:     ϒ(α) := ϒ(α) − 1;
16:     break;
17:   end for
18:   if runable = false then
19:     return false;
20:   end if
21: end while
22: return true;
```

number of tokens on any channel to be negative. A class S algorithm is an algorithm that schedules an actor if it is runnable, updates the number of tokens on channels and terminates only when no more actors are runnable. The scheduling process is deadlocked if it terminates before each actor has been scheduled the number of times specified in the $\mathbf{\Upsilon}$ vector.

Theorem 3 ([LM87b])**.** Given the topology matrix $\mathbf{\Gamma}$ of an SDFG, if a positive integer vector $\mathbf{\Upsilon}$ satisfies $\mathbf{\Gamma} \times \mathbf{\Upsilon} = \mathbf{0}$, then any class S algorithm will find a legal periodic schedule for the SDFG if such a schedule exists.

Based on the sufficient condition in Theorem 3, a class S algorithm can be used to check if a legal periodic schedule exists for an SDFG. A deadlock checking algorithm

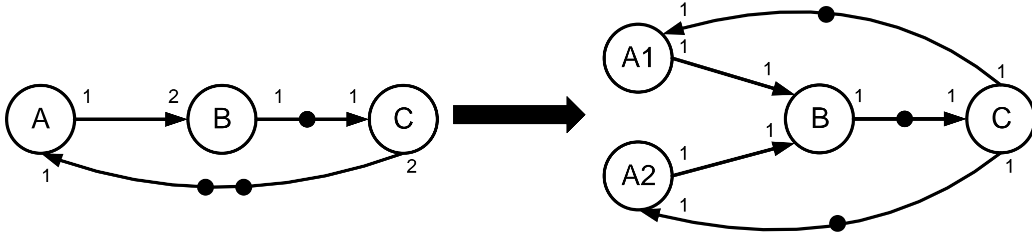


Figure 2.2: An SDFG-to-HSDFG conversion example.

based on a class S algorithm proposed in [LM87b] is given in Algorithm 1. If the scheduling process in Algorithm 1 is not deadlocked for an SDFG, then at least one legal periodic schedule can be found for the SDFG.

If at least one legal periodic schedule exists for an SDFG, the SDFG is referred to as a **legal SDFG**. All SDFGs to be scheduled in this thesis are assumed to be legal SDFGs by default.

2.3.4 SDFG-to-HSDFG Conversion

Every SDFG has an equivalent HSDFG. The schedules derived from the equivalent HSDFG are applicable to the original SDFG. Since HSDFGs are much easier to be handled, SDFG-to-HSDFG conversion is widely used in SDFG scheduling as a preprocessing step. On the other hand, this conversion also brings about a problem that, for the obtained graph, an exponential increase of the number of the actors may happen after the conversion [Stu07, PBL$^+$95, SB09]. A conversion algorithm is provided in Section 3.8 of the book [SB09]. In this thesis, the equivalent HSDFG of an SDFG $G=(V,E)$ is denoted by $G_H=(V_H,E_H)$.

An example of a conversion from an SDFG to its equivalent HSDFG is shown in Figure 2.2, where actors $A1$ and $A2$ in the HSDFG are two duplications of actor A in the original SDFG.

2.3.5 Iteration Period and Throughput

The definition of **iteration period** (IP) and **throughput** of an SDFG is given below.

Definition 12 (Iteration Period and Throughput)**.** For a periodic schedule of an SDFG $G=(V,E)$, its iteration period IP is the time consumed to execute all actors $\alpha \in V$ $\Upsilon(\alpha)$

times, where Υ is the repetition vector of the SDFG. Then, $TH = 1/IP$ is the throughput of the schedule.

There is an inherent **lower bound** (LB) for the IP of an SDFG with **cycles**. Here is the definition of a cycle in an SDFG.

Definition 13 (Cycle)**.** A cycle CC in an SDFG $G = (V, E)$ is a path $(e_1, e_2, ...e_n)$ in the SDFG, where $\alpha_{src}(e_1) = \alpha_{dst}(e_n)$.

An SDFG with cycles is referred to as **cyclic SDFG**. Since there are multiple token producing/consuming rates in an SDFG, it is difficult to express in a straightforward manner the LB for the IP of a cyclic SDFG. Thus, the LB of an SDFG is always derived from its equivalent HSDFG. For a deadlock-free HSDFG $G_H = (V_H, E_H)$, the LB of its IP is its maximum cycle mean (MCM) as defined in Equation 2.2 [SB09].

$$MCM(G_H) = \max_{\forall CC \in G_H} \frac{\sum_{e \in CC} ET(\alpha_{dst}(e))}{\sum_{e \in CC} it(e)}, \tag{2.2}$$

where CC represents a cycle in G_H.

Since any SDFG can be converted to an HSDFG, in a traditional way, the LB of an SDFG can always be obtained through the analysis of its equivalent HSDFG. To avoid the SDFG-to-HSDFG conversion process and the explosion of the graph size caused by this conversion, a throughput analysis method is proposed in [GGS$^+$06] which can get the LB (or maximum throughput) of an SDFG directly based on self-timed execution (STE). The STE technique is introduced in detail in Section 2.7.4 in this chapter.

A schedule which has the maximum throughput is referred to as a **rate-optimal** schedule. In other words, a rate-optimal schedule should have an IP equal to LB.

2.4 SDFG Transformation Techniques

In theory, without the constraint of the number of processing cores, a legal SDFG must have a rate-optimal schedule. However, the parallelism within an SDFG may not be easily used without using appropriate SDFG transformation techniques.

2.4.1 Retiming

The scheduling of SDFGs has to be periodic to support the iterative execution of SDFGs. However, a **retiming** [LS91] (or pipelining [LM$^+$87a]) technique adjusts the

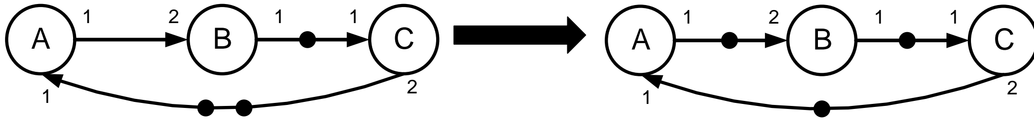
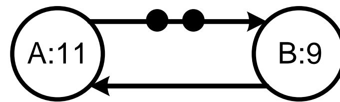


Figure 2.3: A retiming example.

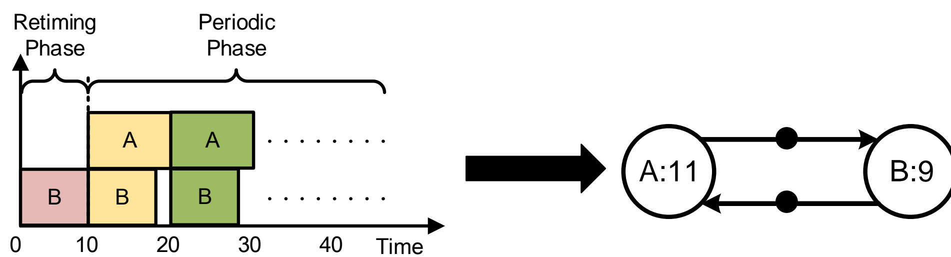
distribution of initial tokens in the original SDFG by conducting a **retiming phase** before the periodic scheduling. In the retiming phase, each actor α fires $RT(\alpha)$ times, where RT is referred to as the **retiming set** of an SDFG. An example is shown in Figure 2.3, in which actor A is retimed for once. This technique is firstly proposed by Leiserson et al. [LRS83, LS91] to redistribute registers in synchronous circuitry. A synchronous circuitry can be described as a dataflow graph; then the registers in the circuitry are equivalent to the initial tokens in the dataflow graph.

A retiming process, which enables some actors to be executed before the periodic scheduling, turns some intra-iteration data dependencies into inter-iteration data dependencies. The inter-iteration data dependencies do not matter when scheduling one iteration of an SDFG, which means some data dependencies in the original SDFG are eliminated by the retiming process. From another perspective, the retiming process overlaps the execution of actors in different iterations by letting some actors execute in advance of the periodic execution phase and makes the utilization of the inter-iteration parallelism to be possible.

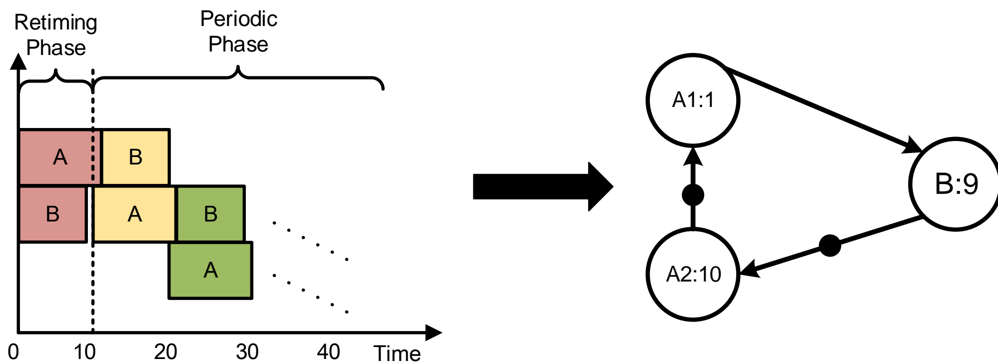
Except for the **classic retiming** approaches which redistribute the initial tokens of an SDFG, a more generalized retiming or pipelining process also enables partial execution of an actor to be performed in the retiming phase. This kind of retiming is denoted as **general retiming** in this thesis. Since the classic retiming is included in the general retiming, the general retiming holds more advantages than the classic retiming in terms of throughput improvement for SDFGs. For example, for the HSDFG shown in Figure 2.4a where the numbers in actors represent the execution time of the corresponding actors, a classic retiming can only achieve a periodic schedule with an IP of 11, which is restricted by the execution time of actor A. The equivalent transformation of the retiming process for the original HSDFG is shown on the right side of Figure 2.4b. The general retiming shown in Figure 2.4b can achieve an optimal schedule with an IP of 10. As we can see on the right side of Figure 2.4c, the general retiming splits the actor A into two actors $A1$ and $A2$. This transformation is fulfilled by executing actor A ten time units in the retiming phase as shown on the left side of Figure 2.4c.



(a) An example HSDFG for general retiming.



(b) A classic retiming of the example HSDFG.



(c) A general retiming of the example HSDFG.

Figure 2.4: The classic retiming and the general retiming of an example HSDFG.

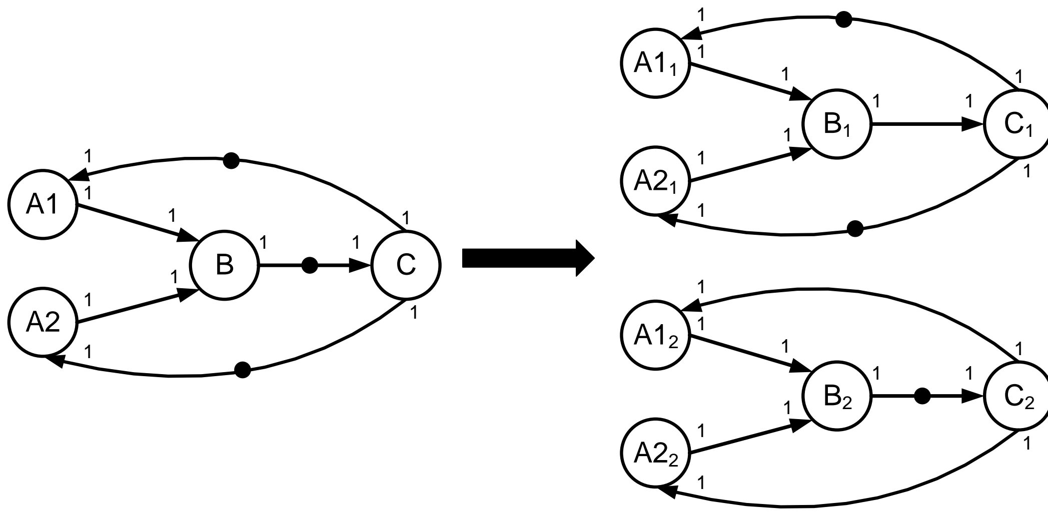


Figure 2.5: An unfolding example.

2.4.2 Unfolding

A dataflow graph usually describes one iteration of an iterative application. The technique of **unfolding** is used to unfold the original graph and make more iterations to be explicitly revealed in one iteration of the unfolded graph, so that the parallelism between different iterations can be presented in the unfolded graph and utilized in the following scheduling process. If an unfolded graph describes J consecutive iterations of the original graph, then the unfolded graph is referred to as the J-unfolded graph, where J is called the **unfolding factor**. If the IP of a schedule for the unfolded SDFG is IP_J, where J is the unfolding factor, then the average IP of one iteration of the original SDFG is IP_J/J and the throughput of the schedule is J/IP_J. An unfolding for an example HSDFG is shown in Figure 2.5, in which the unfolding factor for the original graph is 2. In Figure 2.5, the HSDFG to be unfolded is the HSDFG obtained from the SDFG-to-HSDFG conversion shown in Figure 2.2. Since the unfolding factor is 2, all the actors in the original HSDFG have been replicated twice in the unfolded HSDFG. For example, actor $A1$ in the original HSDFG has two replications $A1_1$ and $A1_2$ in the unfolded HSDFG.

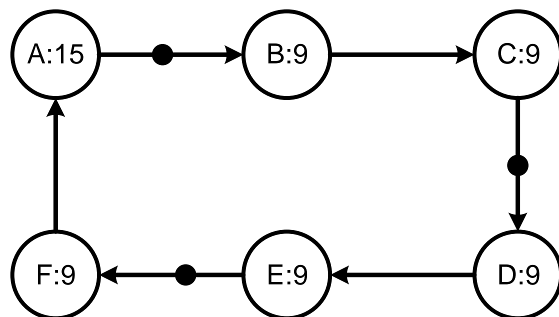
For a given unfolding factor, an unfolding algorithm for HSDFGs is firstly provided by Parhi and Messerschmitt in [PM91]. Then, a more succinct unfolding algorithm is proposed by Chao and Sha in [CS97] as shown in Algorithm 2.

Algorithm 2 Unfolding Algorithm [CS97]

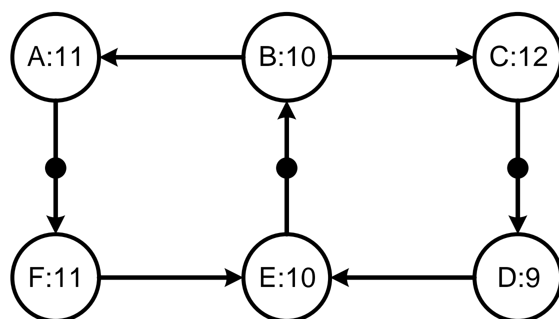
Input:
An HSDFG $G = (V, E)$; Unfolded factor J;
Output:
The unfolded HSDFG $G_J = (V_J, E_J)$;
Iteration:
1: **for all** $\alpha \in A$ **do**
2: Add J copies of α to the actor set A_J of G_J, which are denoted as $\alpha_1, \alpha_2, ..., \alpha_J$;
3: **end for**
4: **for all** $e \in E$ **do**
5: $\delta := it(e) \mod J$; $\rho := \lfloor it(e)/J \rfloor$;
6: **for all** Integer i in range $[1, J - \delta]$ **do**
7: Add an edge $e' = \alpha_{src}(e)_i \rightarrow \alpha_{dst}(e)_{i+\delta}$ to edge set E_J, where $it(e') = \rho$;
8: **end for**
9: **for all** Integer i in range $[J - \delta + 1, J]$ **do**
10: Add edge $e' = \alpha_{src}(e)_i \rightarrow \alpha_{dst}(e)_{i+\delta-J}$ to edge set E_J, where $it(e') = \rho + 1$;
11: **end for**
12: **end for**

The **optimal unfolding factor** for an SDFG is the smallest unfolding factor that enables the existence of a rate-optimal schedule for one iteration of the unfolded SDFG. One problem from applying the unfolding technique is that obtaining the optimal unfolding factor for a cyclic SDFG (or even for a cyclic HSDFG) is still an open problem if only classic retiming is used. Yang and Fu [YF97] propose an equation to calculate a near optimization unfolding factor for cyclic HSDFGs. Parhi and Messerschmitt try to give an upper bound of the optimal unfolding factor in [PM91]. However, the conclusion of Parhi and Messerschmitt is disproved by Wolz and Kolla in [WK03]. Wang and Hu propose a necessary and sufficient condition to evaluate if an unfolding factor is optimal or not in [WH92] for cyclic HSDFGs. After that, they twice overturn their previous conclusions and propose modified necessary and sufficient conditions in their two following papers [WH94a] and [WH94b]. However, even the latest necessary and sufficient condition they propose in [WH94b] is still incorrect. Besides, an algorithm proposed in [LP93] claims to be able to get the optimal unfolding factor for HSDFGs. However, this work can also be easily proven incorrect by a counterexample.

The unfolding factors (or collapsing factors) in [WH92, WH94a, WH94b] are obtained based on Theorem 3 in [WH92], Theorem 1 in [WH94a], and Theorem 1 in [WH94b], respectively. However, these theorems can be disproved by the two counterexamples as shown in Figure 2.6. The two graphs in Figure 2.6 are both HSDFGs.



(a) A counterexample HSDFG.



(b) Another counterexample HSDFG.

Figure 2.6: Two counterexamples.

The letter and the number shown inside each actor are the label and the execution time of the corresponding actor, respectively. For the first counter case in Figure 2.6a, the HSDFG is a Generalized Perfect Rate Graph (GPRG) or Modified Generalized Perfect Rate Graph (MGPRG) as defined in papers [WH92, WH94a, WH94b], which means the HSDFG has at least one schedule that makes the IP of the graph equals to its LB. For one iteration of a cyclic HSDFG, a **critical path** is a path with the maximum length among the paths without containing any edges with initial tokens in the HSDFG. The IP of one iteration of a cyclic HSDFG is decided by the length of its critical path. For the HSDFG in Figure 2.6a, the LB (or MCM) is $60/3 = 20$. However, without unfolding, the shortest IP of the HSDFG is 24 (i.e. the length of the critical path $F \to A$, or $A \to B$ if B, D and F are all retimed once).

Another counterexample is shown in Figure 2.6b, which is also a GPRG or MGPRG in [WH92, WH94a, WH94b]. In the HSDFG in Figure 2.6b, the LB of the graph is 21. There are two cycles in the graph, sharing a common edge $E \to B$. This shared edge causes a problem as when we try to make the critical path of the left cycle to

be equal to the LB using retiming, the critical path of the right cycle has to become longer than the LB, and vice-versa. According to these counterexamples, the LB of a cyclic HSDFG is not decided by the longest execution time of its actors as argued in [WH92, WH94a, WH94b], but by the length of the critical path in the HSDFG. The condition that LB multiplied by the unfolding factor should be equal to or larger than the longest execution time of the actors is necessary but not sufficient to guarantee an HSDFG to have a rate-optimal schedule (i.e. the *Only If* parts of the proofs for the theorems in [WH92, WH94a, WH94b] are correct, yet the If parts are incomplete). A correct, necessary and sufficient condition for the existence of a rate-optimal schedule for one iteration of a cyclic HSDFG is given in this thesis as shown in Theorem 4.

Theorem 4. One iteration of a cyclic HSDFG has a rate-optimal schedule if and only if there exists a retiming transformation for the cyclic HSDFG that makes the length of the critical path in the retimed HSDFG to be equal to or less than LB.

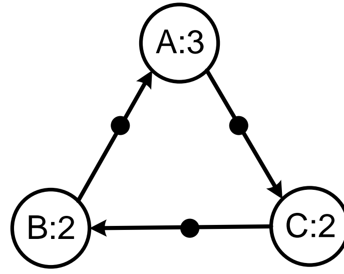
According to Theorem 4, a lower bound for the optimal unfolding factor of a cyclic HSDFG is

$$J_L = \left\lceil \frac{L_{P_{min}}}{LB} \right\rceil, \tag{2.3}$$

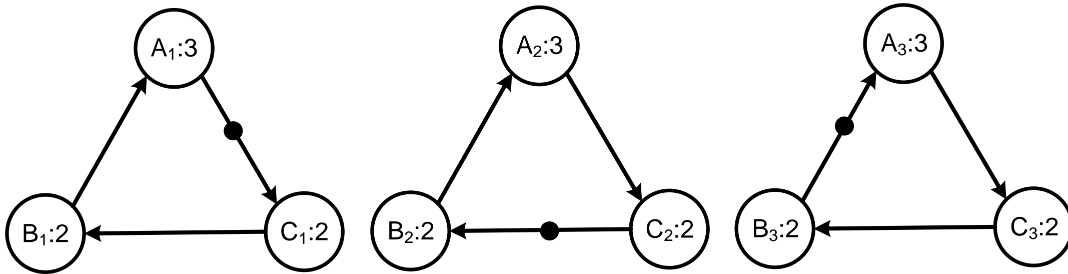
where $L_{P_{min}}$ is the minimum length of the critical path for all the possible retimed HSDFGs of the original HSDFG. On the other hand, obtaining a higher bound for the optimal unfolding factor of an HSDFG is still an open problem since the higher bound proposed in [PM91] has been disproved in [WK03].

An algorithm to get the optimal unfolding factor of an HSDFG is proposed in [LP93]. The algorithm keeps unfolding an HSDFG using J_L of the HSDFG and making the unfolded HSDFG as a new HSDFG to be unfolded, until the value of J_L equals to 1. However, this algorithm cannot always find the optimal unfolding factor. The HSDFG in Figure 2.6a can be a counterexample. The optimal unfolding factor of the HSDFG is 3. However, according to the algorithm in [LP93], $J_L = 2$ for the original HSDFG and all the following unfolded HSDFGs. Therefore, only multiples of two can be examined by the algorithm and the real optimal unfolding factor 3 can never be found by the algorithm.

As all existing work claiming to be able to find the optimal unfolding factor has limitations, how to get the optimal unfolding factor for an HSDFG is, to the best of the author's knowledge, still an open problem if only classic retiming is used. On the other hand, if general retiming (as defined in Section 2.4.1) is adopted, the optimal unfolding



(a) An example HSDFG.



(b) The unfolded HSDFG using the optimal unfolding factor 3.

Figure 2.7: An examples to illustrate that unfolding an HSDFG by its optimal unfolding factor may lead to sub-optimal throughput under the constraint of a fixed number of processing cores.

factor can be simply obtained by Equation 2.4 proposed in [CS92].

$$J_{opt} = ET(C_{cr})/gcd(ET(C_{cr}), it(C_{cr})) \tag{2.4}$$

Note that the discussion above is only for HSDFGs. It is even harder to get the optimal unfolding factor directly from an SDFG since the multiple token producing/consuming rates of actors make the analysis for SDFGs to be more complicated than HSDFGs. For example, no existing work has found or even given a definition of the critical path of an SDFG with multiple rates.

In addition, the optimal unfolding factor may not be the best unfolding factor if the number of processing cores is fixed. For example, Figure 2.7a shows an example HSDFG, which has an optimal unfolding factor of 3. The unfolded HSDFG for the example HSDFG using its optimal unfolding factor is shown in Figure 2.7b. If we schedule the unfolded HSDFG onto a system with two processing cores, then an optimal schedule is to assign two disconnected cycles on the same core and assign the third

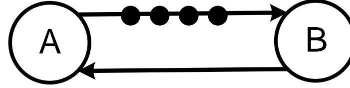


Figure 2.8: A counterexample for using retiming alone.

cycle on the other core. The average IP of one iteration obtained by this schedule is $14/3 \approx 4.7$. For the original HSDFG in Figure 2.7a, an optimal schedule for a system with two processing cores is to schedule actors *B* and *C* on the same core and schedule actor *A* on the other core. The IP of this schedule is 4, which is shorter than the average IP of the optimal schedule for the unfolded HSDFG in Figure 2.7b. Therefore, using the optimal unfolding factor to unfold an SDFG does not always lead to a better unfolded SDFG for scheduling under the constraint of the number of processing cores. How to get the best unfolding factor under the constraint of the number of processing cores remains an open problem.

2.4.3 Combinational Use of Retiming and Unfolding

Both retiming and unfolding can improve the parallelism of SDFGs. These two techniques cannot replace each other as stated in the following theorem.

Theorem 5. Using retiming or unfolding alone cannot obtain rate-optimal schedules for all SDFGs.

Proof. To prove Theorem 5, we only need to give two counterexamples which cannot get rate-optimal schedules when only retiming or unfolding is used.

A counterexample for using retiming alone is given in Figure 2.8, where actors *A* and *B* have the same execution time τ ($\tau > 0$) and the token producing and consuming rates of *A* and *B* are 1. The LB of the SDFG in Figure 2.8 is $\tau/2$. If only retiming is used, the best retiming is to retime actor *B* once. Then, the IP of the optimal schedule of the retimed SDFG is τ, which is decided by the execution time of actor A or B. The IP τ is larger than the LB of the original SDFG $\tau/2$. Therefore, no rate-optimal schedule exists for the optimal retiming of the SDFG in Figure 2.8 if only retiming is used. For general retiming, this conclusion still holds for any τ that is not divisible by 2 according to Equation 2.4.

A counterexample for using unfolding alone is shown in Figure 2.9, where actors

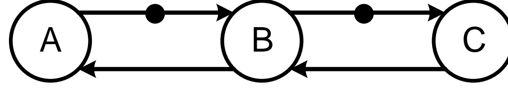


Figure 2.9: A counterexample for using unfolding alone.

A, B and C have the same execution time τ ($\tau > 0$) and the token producing and consuming rates of all actors are 1. The LB of the SDFG in Figure 2.9 is 2τ. The critical path in the original SDFG is $C \rightarrow B \rightarrow A$. Therefore, without retiming, the IP of the optimal schedule for the original SDFG is 3τ. If only the unfolding technique is considered and an integer J ($J > 1$) denotes the unfolding factor, then unfolding the original SDFG using an arbitrary value of J, there always exists a critical path $C_1 \rightarrow B_1 \rightarrow A_1 \rightarrow B_2 \rightarrow A_2 \rightarrow \ldots \rightarrow B_J \rightarrow A_J$ in the unfolded SDFG. The IP of the optimal schedule for the unfolded SDFG is decided by the length of the critical path which is $(2J+1)\tau$. Then, the average IP of the optimal schedule for one iteration of the original SDFG in Figure 2.9 is $2\tau + \tau/J$, which is always longer than the value 2τ of the LB of the original SDFG. Thus, only using unfolding for the SDFG in Figure 2.9 cannot make any of its unfolded SDFG to have at least one rate-optimal schedule. □

Based on Theorem 5, retiming and unfolding are both essential for getting rate-optimal schedules of an SDFG. Chao and Sha prove that the order of unfolding and retiming does not affect the optimal achievable IP of the transformed HSDFG as shown in Theorem 1 of the paper [CS97]. According to this property, a retiming algorithm is also proposed in [CS97], which performs retiming before unfolding to avoid making the retiming decisions on the unfolded graph. The problem scale of retiming would be smaller on the original graph than the unfolded graph as the size of the original graph has not been multiplied by the unfolding process.

2.5 Problem Models

In this section, the models of the scheduling problems for SDFGs are illustrated. We use $MD(X|Y)$ to denote a model, where X is a list of objectives of the scheduling and Y is a list of constraints for the scheduling. A common constraint for all scheduling problem models is to be deadlock-free, which is denoted as F in this thesis.

2.5.1 Ideal Model

The simplest problem model is the **ideal model**, which assumes the communication overhead between different processors is negligible and the computing resources are infinite, e.g. the scheduling for timed event graphs in [Ram74] and dataflow graphs in [PM91]. The model is suitable for the situations in the system design stage that the hardware resources are not fixed. In addition, the model can also be used to get schedules for SDFGs under certain constraints since many constraints can be modelled by adding extra actors and edges to the original SDFGs. The modelling approaches are detailed in Section 2.6. We denote the ideal model as $MD(X|F)$, where X is the objective list of the scheduling and deadlock-free is the only constraint. Considering that only non-preemptive scheduling is used in this thesis, the model has the following properties [Sin07]:

- For an SDFG $G = (V, E)$, an actor is executable only if $\forall e \in IN(\alpha)$, $b(e) \geq q(e)$, where $b(e)$ is the number of tokens on channel e;
- The execution of actors cannot be interrupted, which means the difference of the finishing time of the execution of an actor and the starting time of the execution of the actor is exactly the execution time of the actor.

2.5.2 The Number of Processing Cores

For systems which consist of homogeneous processors while the inter-processor communication costs are ignorable (e.g., homogeneous multicore systems with high-speed inter-processor connection or applications with small size of communication data), the number of processing cores should be taken into consideration as an extra constraint compared to the ideal model. The model with the constraint of the number of processing cores is denoted as $MD(X|F,N)$, where N is the number of processing cores. Other than the two properties of the ideal model, the model $MD(X|F,N)$ has two additional properties:

- The execution time of the actors on the same core cannot be overlapped with each other;
- The number of concurrently firing actors cannot be greater than the number of processors.

In the design stage of a system or to improve the utilization of computing resources, minimizing the number of used cores can also be an objective of scheduling. In this case, the model is denoted as $MD(N|F)$.

2.5.3 Communication Architecture

To model a system more precisely, the communication architecture of the system and inter-core communication overhead should be taken into consideration during scheduling. We denote this kind of scheduling as **communication-aware scheduling**. In this case, both data locality and load balancing decide the throughput of a multicore system. Since data locality and load balancing may conflict with each other, the model becomes more complicated when the communication overhead is considered. Besides, the complexity of communication-aware scheduling is even improved by the variety of the interconnection architectures of multicore systems, e.g. fully connected interconnections, bus-based interconnections, network-on-chips (NoCs), etc. The communication tasks need to be scheduled on communication resources. The contention of resources should be taken care of in practical interconnection architectures such as bus-based and NoC based interconnections. Furthermore, if the I/O operation of a processor is controlled by an independent unit other than the main computing unit of the processor, which means the computation and the communication in a system can overlap with each other, then how to use such overlapping to eliminate the communication overhead and get higher throughput becomes an attractive problem.

Communication-aware scheduling always targets fixed systems which means the number of cores is known as a constraint for the scheduling. Therefore, the communication-aware scheduling model is denoted by $MD(X|F,N,C)$, where C is the model of the communication architecture in a system. Compared with the $MD(X|F,N)$ model, this model has an additional property.

- At a time point, an actor can be executed on a core only if all the data required by the execution of the actor has been received by this core before this time point.

2.5.4 Heterogeneity of Processors

In a more generic model, the processors in a system can be heterogeneous. Recently, the heterogeneity of the processing cores on embedded multicore systems is more and more appreciated. For example, the collocation of high performance processing cores, high

efficiency processing cores, GPUs and various customized accelerators (e.g., image signal processor, video decoder, etc.) is common on CPUs in smart phones [SJ14].

There are two categories of heterogeneity. One category is called **weak heterogeneity**, which represents systems with processors which share a same architecture, but the clock frequency, memory size or other resources are different. Identical processors which support dynamic voltage and frequency scaling (DVFS) technique [SMB+02] can also form this type of heterogeneity when different degrees of DVFS are conducted on different processors. For systems with weak heterogeneity, any actor can be allocated on any processing cores while the execution time of an actor can be different on different cores.

The other category is called **strong heterogeneity**, which represents systems with totally different processing elements. For example, a system consists of processors, GPUs, FPGAs or customized accelerators for certain computing tasks. For systems with strong heterogeneity, a processing core may not be able to support the execution of all actors, and a customized accelerator may provide remarkable speedups compared with a generic computing unit for certain computing tasks.

Generally, the performance and energy consumption of heterogeneous systems are better than homogeneous systems, while the design difficulty of these systems is just the opposite [KTJR05, Tei12]. The scheduling model for a heterogeneous system is denoted by $MD(X|F,N,H)$, where H denotes that heterogeneous processing cores are used.

2.5.5 Scheduling Objectives

Throughput

The throughput of a streaming system is always one of the most important design criteria. The scheduling model for optimizing throughput is denoted as $MD(TH|F)$, where TH is the throughput. The optimal schedules for $MD(TH|F)$ can be obtained by solving a series of linear inequalities as shown in [HM93]. The linear inequalities are proven to be transformable to a shortest path problem in [Ram74], which can be solved by Bellman-Ford algorithm [Bel58] in polynomial time. The linear inequalities also form a linear programming problem which can be solved using existing solvers.

However, these two solutions do not consider the unfolding of the original SDFG or HSDFG. Therefore, the achieved throughput by these solutions may not be optimal in some cases if unfolding is not used, e.g. the HSDFG in Figure 2.8. When the number

of the processing cores is used as a constraint, the model is denoted as $MD(TH|F,N)$. This scheduling problem becomes an NP-complete problem even without considering the retiming and unfolding. For HSDFGs, a proof of the NP-completeness for $MD(TH|F,N)$ is given in [DGGH92]. Simply, an SDFG can be converted to a DAG, and the scheduling problem for a DAG under a certain number of processing cores is a well-known NP-complete problem. Therefore, the scheduling problem for model $MD(TH|F,N)$ is also an NP-complete problem. The throughput may also be used as a constraint for scheduling while optimizing other scheduling objectives.

Code Size

Besides throughput, memory size minimization is also a crucial problem for embedded systems which usually have limited on-chip resources. Some embedded systems do not even have off-chip memory due to the limitation of cost. Furthermore, the accessing time of off-chip memory is always significantly longer than the accessing time of on-chip memory, which degrades the performance of a system and brings extra energy consumption. The memory occupied by a streaming application can be categorized as code memory and data memory.

For the code generation of a streaming application on a single processing core, there are two ways to construct the code of the application from a scheduling result: **inline based** approach and **function based** approach. For a scheduling sequence *ABCBC*, where *A*, *B* and *C* are actors in an SDFG, the inline based approach simply combines the code blocks of *A*, *B* and *C* according to the scheduling sequence. The code blocks of *B* and *C* appear twice in the scheduling sequence and therefore two code blocks of *A* exist in the generated code of the application. On the other hand, the function based approach makes the code block of each actor in an SDFG as a function and calls the corresponding functions according to the scheduling sequence in the generated code of the application. Apparently, the function based approach leads to less code size than the inline based approach. However, function calls are time consuming, especially for applications modelled by SDFGs with fine-grain actors. Therefore, how to minimize the code size for inline based approach on a single processing core is a well studied problem and introduced in Section 2.8.1. This problem model is denoted as $MD(CD|F,N_s)$, where N_s means single core.

For a multicore system, since an actor in an SDFG may execute multiple times within one iteration, the code block of the actor may be duplicated on multiple processing cores. A **binding based** scheduling approach enforces each actor to only be mapped on

one core. However, the binding based approach may significantly limit the performance of scheduling since the data-level parallelism within an actor cannot be utilized. To fully utilize the parallelism in an SDFG on multicore systems, **duplication enabled** scheduling approach allows the multiple execution of an actor to be on different cores. However, for the duplication enabled scheduling approach, how to reduce unnecessary code duplication is still an open problem. This problem is denoted as $MD(CD|F,N)$, where CD is the code size of an SDFG on multiple processing cores.

Buffer Size

The data memory use of an SDFG consists of the memory used by the internal computation of actors and by the data transferred between actors through the buffers on channels. During scheduling, the former is barely affected, while the latter can be optimized. Conservatively, the buffer space on each channel is assumed to be independent from the buffers on other channels and unable to be shared with other buffers. Nevertheless, in some work, the memory sharing of buffers is enabled to reduce the buffer size. In this thesis, the conservative assumption is applied automatically.

As shown in [SGB06a], there is a trade-off between throughput and buffer size; these two objectives are usually considered simultaneously during scheduling. For rate-optimal scheduling, the buffer size minimization problem for HSDFGs is NP-complete as proven in [MBGS10]. The buffer size minimization problem for rate optimal scheduling is denoted as $MD(B|F,TH_{opt})$, where B is the buffer size and TH_{opt} is the optimal throughput.

Other Scheduling Objectives

In this thesis, the optimization objectives of the proposed scheduling framework are throughput and memory usage on homogeneous and heterogeneous multicore systems. However, there are many other interesting optimization objectives to be studied in future work, e.g. energy consumption, latency, fault-tolerance, etc.

2.6 Analysis and Modelling Approaches

As mentioned in Section 2.3.5, throughput analysis methods for SDFGs are well-developed, e.g. MCM [SB09] for HSDFGs and the self-timed execution based approach

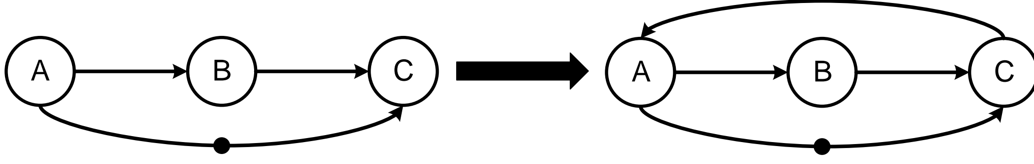


Figure 2.10: A counterexample for buffer bound modelling in [WBJS06] and [SGB06a].

[GGS+06] for SDFGs. A widely used approach to evaluate the throughput of a schedule for an SDFG on a system is to model the schedule by adding extra actors and edges to the original SDFG to construct a **schedule-constrained** SDFG. Then, the throughput of the scheduled-constrained SDFG can be analyzed conveniently using existing approaches. This SDFG transformation based modelling strategy can also be used for the scheduling of an SDFG under certain constraints such as buffer size, auto-concurrencies of actors, etc.

2.6.1 Modelling Buffer Bound

The constraint of buffer size on channels can be modelled by adding a **back-pressure edge** to each channel as shown in [WBJS06] and [SGB06a]. According to [WBJS06] and [SGB06a], for a channel e with a bounded buffer size b_{max}, its back-pressure edge is a channel e', where $\alpha_{src}(e') = \alpha_{dst}(e)$, $\alpha_{dst}(e') = \alpha_{src}(e)$, $p(e') = q(e)$, $q(e') = p(e)$ and $it(e') = b_{max} - it(e)$.

However, using $it(e') = b_{max} - it(e)$ to get the number of initial tokens on a back-pressure edge can lead to unnecessary deadlocks for an SDFG. For example, the HSDFG on the left side of Figure 2.10 has a channel $A \to C$ with one initial token. If the upper bound of the buffer size on channel $A \to C$ is 1, then a back-pressure edge $C \to A$ with zero initial tokens should be added to the original HSDFG to model the constraint of the buffer size on channel $A \to C$, as shown on the right side of Figure 2.10. However, the additional edge $C \to A$ and the original channels $A \to B$ and $B \to C$ form a cycle without any initial tokens, causing the buffer constrained HSDFG to be deadlocked.

The reason of such a deadlock is that an initial token is viewed as a real token by the modelling approaches in [WBJS06] and [SGB06a]. However, initial tokens are always used to represent inter-iteration data dependencies between actors and do not necessarily occupy real buffer space. In this case, the deadlock in Figure 2.10 is unnecessary. In [ZGBS16, ZGBS14], a buffer size modelling approach is proposed for

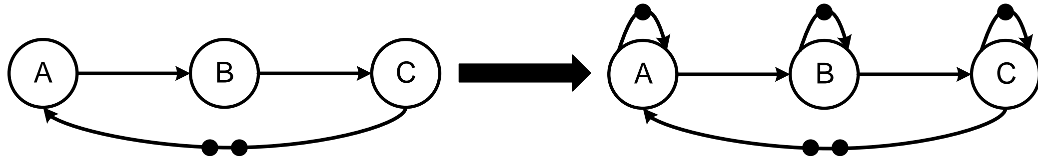


Figure 2.11: An example for modelling auto-concurrences using self-loop channels.

Self-Timed Execution (STE) based scheduling, which avoids the flaw of the modelling approaches in [WBJS06] and [SGB06a]. The constraint of buffer size is modelled as a part of the state variables during the STE scheduling process. The details of STE based scheduling are given in Section 2.7.4.

2.6.2 Modelling Auto-Concurrencies

The **auto-concurrency** of an actor constrains how many invocations of an actor can execute concurrently. The maximum number of concurrent invocations of an actor is referred to as the degree of auto-concurrency of the actor. In an SDFG, some actors may not have multiple concurrent invocations. For example, in [FKBS11] the execution of a **stateful** actor depends on its internal state, which is updated by every invocation of the actor. Thus, the invocations of a stateful actor have to be in sequence to guarantee the functional correctness of an SDFG. The auto-concurrency of an actor α can be modelled by adding a self-loop channel (a channel $\alpha \rightarrow \alpha$) for the actor. The number of initial tokens on the self-loop channel of an actor decides the degree of the auto-concurrency of the actor. An example for modelling auto-concurrencies using self-loop channels is shown in Figure 2.11. The original HSDFG is shown on the left side of Figure 2.11; and the auto-concurrency-constrained HSDFG is given on the right side of Figure 2.11, where actors A, B and C are all restricted to have one degree of auto-concurrency.

2.6.3 Modelling Scheduling and Mapping

For HSDFGs, the modelling of the actor-to-core mapping and the scheduling order of actors on the same processing core are straightforward by adding edges to the original HSDFGs. A modelling method is given in [BKKB02]. According to the method, for a series of actors mapped onto the same core in the order $\alpha_1, \alpha_2, ...\alpha_n$, the modelling operates as below.

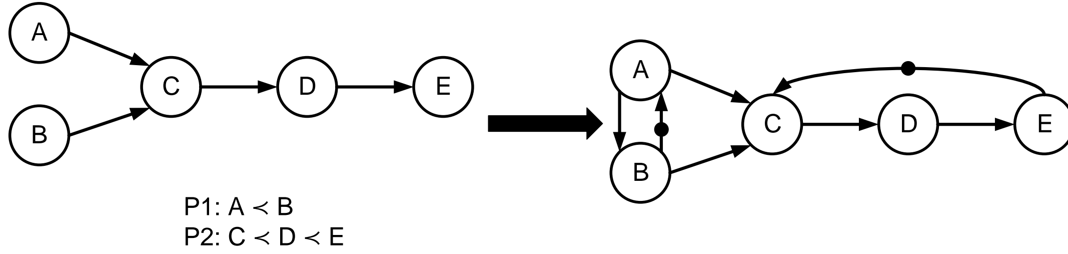


Figure 2.12: An example for modelling scheduling and mapping for an HSDFG.

1. $\forall pair(\alpha_i, \alpha_{i+1}), 1 \leq i < n$, if there are no edges with zero initial tokens connecting α_i and α_{i+1}, an edge $\alpha_i \rightarrow \alpha_{i+1}$ is added to the original HSDFG;

2. Adding an edge $\alpha_n \rightarrow \alpha_1$ with one initial token to the original HSDFG;

3. In the case where $n = 1$, meaning only one actor is mapped on a core, then a self-loop channel with one initial token is added to the actor in the original HSDFG.

An example is shown in Figure 2.12, where an HSDFG and its mapping on two processing cores P1 and P2 are given on the left side. The order of actor execution on the two cores is also given, e.g. $A \prec B$ means the execution of actor A precedes the execution of actor B. The schedule-constrained HSDFG is shown on the right side of Figure 2.12. For core P1, only one pair of actors (A, B) needs to be checked. Since there are no edges connecting actors A and B, two edges $A \rightarrow B$ without initial token and $B \rightarrow A$ with one token are added to the original HSDFG. For core P2, there are two pairs of actors (C, D), (D, E) to be checked. Since edges $C \rightarrow D$ and $D \rightarrow E$ both exist in the original HSDFG and have no initial tokens, only the edge $E \rightarrow C$ with one initial token needs to be added to the original HSDFG.

For generic SDFGs, only the modelling for binding based mappings is supported in existing work. In [DSB+12] and [DSB+13], a decision state modelling technique is proposed to represent a binding based mapping and the schedule of actors on each core. The technique is also adopted in [LF13] to model pipelined scheduling for SDFGs. How to model duplication-enabled mapping using SDFG modification based techniques is still an open problem for generic SDFGs.

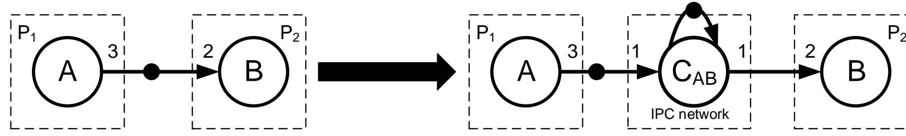


Figure 2.13: An example for modelling communication delays.

2.6.4 Modelling Communication Overhead

The inter-processor communication (IPC) overhead includes two parts: delays for data communication on an IPC network and contention of communication resources such as a shared bus on a bus-based multicore system. For a channel e, if $\alpha_{src}(e)$ and $\alpha_{dst}(e)$ are mapped onto two different processing cores p_1 and p_2 respectively, the IPC delay on the channel e can be modelled as an extra communication actor α_c as shown in [SBGC07]. The modelling process consists of four steps:

1. Adding an actor α_c to the original SDFG while letting the execution time of α_c to be the communication delay of transferring one token on channel e through an IPC network connecting p_1 and p_2;

2. Adding a self-loop channel to α_c with one initial token to ensure the sequential transmission of tokens;

3. Adding two channels $e_s = \alpha_{src}(e) \to \alpha_c$ and $e_r = \alpha_c \to \alpha_{dst}(e)$ to the original SDFG while letting $it(e_s)$=$it(e)$, $p(e_s) = p(e)$, $q(e_s) = p(e_r) = 1$, $q(e_r) = q(e)$, and $TS(e_s) = TS(e_r) = TS(e)$;

4. Removing the channel e from the original SDFG.

An example is shown in Figure 2.13, where actors A and B in an SDFG are allocated on processing core P1 and P2 respectively as shown on the left side of the figure. The communication delay on channel $A \to B$ is modelled as an actor C_{AB} as shown on the right side of Figure 2.13.

In [SBGC07], the initial tokens on the original channel e are transferred onto the sending channel e_s. However, this is not always reasonable as, in a schedule, the starting time of the execution of actor $\alpha_{dst}(e_r)$ may be earlier than the starting time of the communication actor α_c. Therefore, in this thesis, the initial tokens on e_s can be relocated to e_r using retiming. In addition, the IPC network is assumed to transfer one token at a time in [SBGC07], which is not always realistic. The number of tokens to

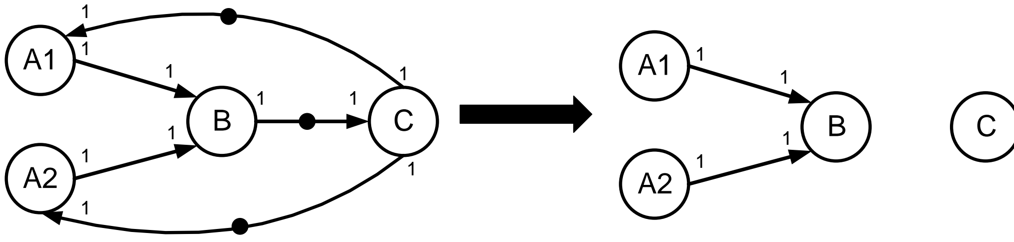


Figure 2.14: HSDFG-to-DAG conversion for an HSDFG.

be transferred by the communication actor α_c can be adjusted by using different token consuming rate $q(e_s)$ and producing rate $p(e_r)$ of α_c. The value of $q(e_s)$ and $p(e_r)$ must be identical and the adjustment range of the value of $q(e_s)$ and $p(e_r)$ is from 1 to $p(e)$.

For the contention of communication resources, since the communication delay on each channel is modelled as a communication actor, the sharing of the communication resources by the communication actors can be viewed as the sharing of a processor by the computing actors mapped on it and be modelled using the approaches discussed in Section 2.6.3.

2.7 Scheduling Methodologies

2.7.1 Blocked Scheduling

Since the channels with initial tokens in HSDFGs only represent data dependencies between iterations, these channels can be removed from the original HSDFG if we only consider data dependencies within one iteration. For any legal HSDFG, there must exist at least one initial token on each cycle in the HSDFG or the HSDFG will be deadlocked. Therefore, after the channels with initial tokens have been removed from a legal HSDFG, the HSDFG becomes a directed acyclic graph (DAG). An HSDFG-to-DAG conversion example is shown in Figure 2.14. Since an SDFG can be converted to its equivalent HSDFG, which can be further converted to a DAG by removing the inter-iteration dependencies, any SDFG can be converted to a DAG if only intra-iteration dependencies are considered.

Based on this property, one scheduling methodology for SDFGs called **blocked scheduling** is proposed in [LM+87a]. The blocked scheduling only schedules J-iteration of an SDFG by converting the SDFG to a DAG which consists of J

iterations of the original SDFG. A detailed conversion algorithm is given in [LM87b]. This conversion is equivalent to unfolding the original SDFG J times and converting the unfolded SDFG to a DAG. As an SDFG is converted to a DAG, scheduling algorithms for DAGs can be used to schedule the converted DAG. Applying the schedule obtained from the converted DAG on every J iterations of the original SDFG, we can get a periodic schedule. The advantage of this scheduling methodology is that it is simple and can make use of well studied DAG scheduling algorithms such as the algorithm proposed in [THW02].

However, an SDFG-to-DAG conversion has to be applied to SDFGs. The conversion is time-consuming and the size of the obtained DAG can be exponentially larger than the original SDFG. Furthermore, to improve the performance of the schedules obtained by the blocked scheduling, explicit retiming and unfolding are always necessary, making the scheduling problem to be more complicated. As shown in [CS97], without the constraint of the number of cores, the optimal retiming of an HSDFG can be found in polynomial time. Nevertheless, the retiming problem can be an NP-complete problem when the number of processing cores and the communication overhead are taken into consideration as shown in [WLW$^+$10]. The same is true for unfolding; obtaining the optimal unfolding factor is still an open problem even for HSDFGs when the number of processing cores is considered.

2.7.2 Solver Based Periodic Scheduling

Besides converting SDFGs to DAGs, another strategy for SDFG scheduling is to construct periodic schedules directly from SDFGs or HSDFGs. The data dependencies and resource limitations can be modelled as constraint formulas or behavioural functions [CT14] and solved by existing solvers.

Linear Programming Based Scheduling

Linear programming (LP) is a well-known optimization method. The constraints of an optimization problem are modelled as a series of linear inequalities. The objective of the optimization problem is to maximize or minimize a linear function. The LP problem is known as a class P problem, which means it can be solved in polynomial time. In terms of the scheduling problems for SDFGs, the model $MD(TH|F)$ can be modelled as an LP problem as $MD(TH|F)$ is also a class P problem.

If all variables of an LP problem are restricted to integers, the LP problem becomes

an integer linear programming (ILP) problem. Especially, if only partial variables are restricted to integers for an LP problem, it is called a mixed integer linear programming (MILP) problem. Both ILP and MILP problems are NP-complete. One advantage of ILP and MILP is that the achievement of exact solutions can be guaranteed if runtime is long enough. The disadvantage of ILP and MILP is all constraints and objectives have to be represented as linear inequalities, which limits the application of ILP and MILP to scheduling problems on complicated systems. The runtime of ILP and MILP is also relatively long compared to heuristics which only search for local optimal solutions. Popular solvers for LP, ILP and MILP include CPLEX [Cor18], Lingo [Sch06], etc.

Model-Checking Based Scheduling

Besides LP, model-checking is another widely used method to solve scheduling problems for SDFGs. The model-checking technique is initially proposed to check if a system runs properly while satisfying certain specifications, e.g. if it is free from deadlocks or other situations that may crash the system. To do this, a system is modelled mathematically as a finite state machine and verified automatically by a model-checker. For scheduling problems for SDFGs, the execution of an SDFG on a system under scheduling constraints is modelled as symbolic state transformations instructed by mathematical formulas. As model-checking can only prove if a schedule exists under some constraints, to optimize a scheduling objective, a searching process should be used with model-checking jointly. Commonly used solvers for model-checking include SPIN [Hol04] and UPPAAL [BDL+06].

Constraint Programming Based Scheduling

Constraint programming (CP) is proposed to solve hard combinational problems including scheduling. The scheduling problem for SDFGs is modelled as constraint formulas such as logical formulas, linear inequalities or even user-defined functions. Thus, CP is similar with LP except that not only linear constraints are supported, making CP to be more flexible than LP. On the other hand, the speed and usability of CP are usually worse than LP. Gecode [Tea06] is a representative constraint programming solver.

SAT Based Scheduling

The boolean satisfiability (SAT) problem aims to find if there is a satisfiable solution for a series of boolean formulas. The variables in a boolean formula can only have boolean values. The SAT problem is a well-known NP-complete problem. Many practical problems can be modelled as SAT problems, e.g. circuit design, automatic theorem proving and also scheduling. Similar to model-checking, a SAT solver can only be used to check if a schedule satisfies certain constraints. Therefore, a SAT solver must be jointly used with a searching process for the optimization of scheduling objectives. An example SAT solver is MiniMaxSAT [HLO08] used for scheduling SDFGs in [LGY15].

2.7.3 Evolutionary Algorithm based Scheduling

An evolutionary algorithm (EA) is a nature-inspired metaheuristic optimization algorithm [Yan10]. A representative EA algorithm is called genetic algorithm (GA), which is inspired by biotic population evolution influenced by the natural selection. EAs are suitable for most optimization problems and adept at getting global optimal solutions. To obtain an optimal solution, a certain number of candidate solutions are generated as a population. Then, an evolutionary process is performed, which in general consists of mutation, crossover, selection and reproduction. The candidate solutions are evaluated by a fitness function. A more fit solution has a larger possibility to be selected to generate the next generation of the population. Commonly used termination conditions include that the evolutionary process has reached a limited number of generations, the fitness of current solutions has met the target objective, and the upper bound of the runtime has been exceeded.

2.7.4 Self-Timed Execution Based Scheduling

In addition to blocked scheduling and generic optimization algorithms such as LP, model-checking, CP, SAT and EA, a **self-timed execution** (STE) based scheduling approach is proposed specifically for SDFGs in [GGS$^+$06] and further developed in the following work [SGB06a, SBGC07, ZGBS14, ZGBS12].

STE based scheduling simulates the execution process of an SDFG using transformations of symbolic states until a periodic execution pattern has been found. The transformation of symbolic states forms a finite state machine. Thus, the idea of STE is actually similar with model-checking. A **state** in STE represents an execution stage of an SDFG. Analysis and scheduling for an SDFG are conducted by analyzing

and constraining the transition of the states. The definition of a state is given in Definition 14.

Definition 14. For an SDFG $G = (V, E)$, a state in STE of the SDFG is represented by two vectors $\boldsymbol{T}$ and $\boldsymbol{R}$. $\boldsymbol{T}$ is a $|E|$-dimensional vector, where $|E|$ is the number of channels in an SDFG. Each element in $\boldsymbol{T}$ represents the amount of the tokens on a channel. For a channel e, the initial value of $\boldsymbol{T}(e)$ equals to the number of the initial tokens on the channel $e(it)$. $\boldsymbol{R}$ is a $|V|$-dimensional vector, where $|V|$ is the number of actors. The elements in $\boldsymbol{R}$ denote the remaining time of the execution of actors. Since an actor may execute concurrently on multiple processing cores, the elements in $\boldsymbol{R}$ are FIFO queues. Since the execution of actors has not started yet at the beginning of the STE process, the initial values of all elements in R are all empty queues denoted by $\{\}$.

In existing work, the STE based scheduling follows the rule of earliest start time scheduling, which means the actors execute as soon as possible (ASAP) within the STE process. The STE of an SDFG stops when two identical states are found. The states between the two identical states compose a periodic execution pattern, which is referred to as the **periodic phase** of STE. The state transformations before the periodic phase are called the **transient phase** of STE. The STE of an SDFG may not always have a transient phase.

An algorithm in [ZGBS12] to get the STE of an SDFG is given in Algorithm 3. The STE process is output as a list of states *Seq*, where the order of the states in *Seq* follows the sequence of the state transformations during the STE process. After the initialization in lines 1-2, the while-loop (lines 3-20) in Algorithm 3 conducts the STE process, where five operational functions are used, including

- $endReady(\alpha, CurState)$,
- $end(\alpha, CurState)$,
- $startReady(\alpha, CurState)$,
- $start(\alpha, CurState)$,
- $clockProceeding(CurState)$.

Functions $endReady(\alpha, CurState)$ and $startReady(\alpha, CurState)$ are used to check if an actor α is ready to end or start for the current state. An actor is ready to end if and

Algorithm 3 Self-Timed Execution [ZGBS12]

Input:
An SDFG $G = (V, E)$;
Output:
The STE process of the input SDFG represented by a state list *Seq* and the start state of the periodic phase;
Iteration:

```
1:  Initialize the current state CurState according to Definition 14;
2:  Initialize the state list Seq as an empty list;
3:  while True do
4:      for all α ∈ V do
5:          while endReady(α, CurState) do
6:              end(α, CurState);
7:              if CurState ∈ Seq then
8:                  return [Seq, CurState];
9:              else
10:                 Insert CurState to the end of Seq;
11:             end if
12:         end while
13:     end for
14:     for all α ∈ V do
15:         while startReady(α, CurState) do
16:             start(α, CurState);
17:         end while
18:     end for
19:     clockProceeding(CurStates);
20: end while
```

only if the remaining execution time of the actor is 0. The definition of the function $endReady(\alpha, CurState)$ is shown in Equation 2.5.

$$endReady(\alpha, CurState) := \begin{cases} True, & if\ CurState.\boldsymbol{R}(\alpha)_0 = 0; \\ False, & otherwise; \end{cases} \tag{2.5}$$

where $CurState.\boldsymbol{R}(\alpha)_0$ is the first element in the queue $\boldsymbol{R}(\alpha)$.

An actor is ready to start if and only if all input channels of the actor have enough tokens for one invocation of the actor. The definition of the function

$startReady(\alpha, CurState)$ is shown in Equation 2.6.

$$startReady(\alpha, CurState) := \begin{cases} True, & if\ \forall e \in INPUT(\alpha), CurState.\boldsymbol{T}(e) \geq q(e); \\ False, & otherwise. \end{cases} \tag{2.6}$$

The functions $end(\alpha, CurState)$ and $start(\alpha, CurState)$ change the current state by ending or starting the execution of an actor. The actor ending operation is defined as

$$\begin{aligned} end(\alpha, CurState) \equiv_{def} & \big(\forall e \in OUTPUT(\alpha), CurState.\boldsymbol{T}(e) := CurState.\boldsymbol{T}(e) + p(e)\big) \\ & \wedge \big(CurState.\boldsymbol{R}(\alpha).pop()\big), \end{aligned} \tag{2.7}$$

where $CurState.\boldsymbol{R}(\alpha).pop()$ means popping out the first element from the FIFO queue $CurState.\boldsymbol{R}(\alpha)$.

The actor starting operation is defined as

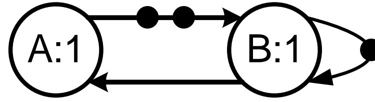
$$\begin{aligned} start(\alpha, CurState) \equiv_{def} & \big(\forall e \in INPUT(\alpha), CurState.\boldsymbol{T}(e) := CurState.\boldsymbol{T}(e) - q(e)\big) \\ & \wedge \big(CurState.\boldsymbol{R}(\alpha).push(ET(\alpha))\big), \end{aligned} \tag{2.8}$$

where $CurState.\boldsymbol{R}(\alpha).push(ET(\alpha))$ means pushing the execution time of α into the end of the FIFO queue $CurState.\boldsymbol{R}(\alpha)$.

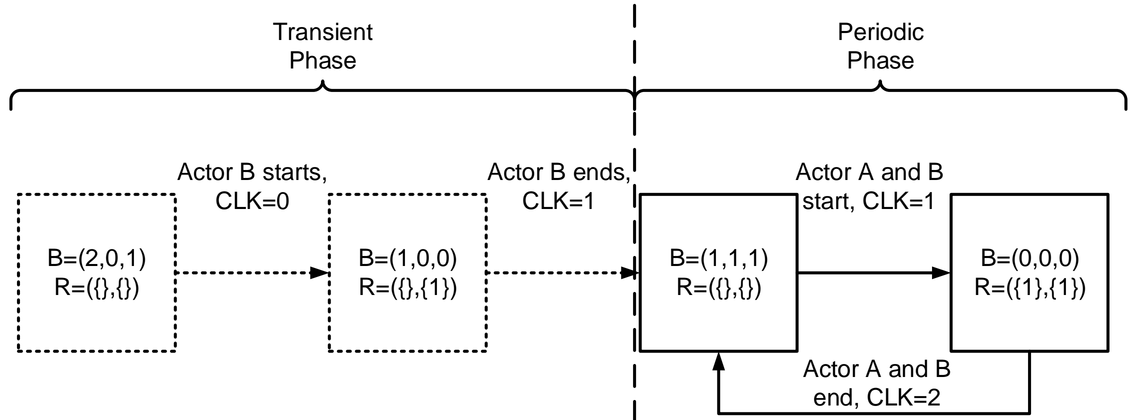
The for-loop between lines 4 and 13 in Algorithm 3 checks all end-ready actors in V and ends the execution of these actors. Each time the execution of an actor has ended, the algorithm checks if an identical state of the updated *CurState* exists in *Seq*. If such a state exists, a periodic execution pattern has been found and the algorithm can terminate successfully by returning *Seq* as the state sequence of the STE and *CurState* as the start state of the periodic phase of the STE. Otherwise, the updated *CurState* is inserted to the end of *Seq* and the STE process goes on.

After the execution of all the end-ready actors have been ended, the for-loop between line 14-18 is used to find all the start-ready actors and start their execution. Then, the function $clockProceeding(CurState)$ advances the STE process until at least one actor becomes ready to end. Thus, the time step should be

$$Step = \min_{\forall \alpha \in V} CurState.\boldsymbol{R}(\alpha)_0. \tag{2.9}$$



(a) An example SDFG for STE.



(b) The STE of the example SDFG.

Figure 2.15: An example SDFG and its STE.

Then, the clock proceeding operation is defined as

$$clockProceeding(CurState) \equiv_{def} \forall r \in CurState.\boldsymbol{R}(\alpha)|\forall\alpha \in V, r := r - Step. \quad (2.10)$$

As an example, Figure 2.15b shows the STE of the SDFG in Figure 2.15a. For the SDFG in Figure 2.15a, the execution time of both actors A and B is 1 time unit, and the data producing and consuming rates on channels are all 1. The STE of an SDFG is represented by the transformation of the states. In Figure 2.15b, the rectangles represent the states of the STE for the SDFG in Figure 2.15a. The leftmost rectangle is the initial state. The states are transformed by the actions shown between two states. The action of actor start consumes tokens on the input channels of the corresponding actor. Similarly, the actor end action produces tokens on the output channels of the corresponding actor. *CLK* shows the time points of the happening of executing operations, e.g. the starting time and ending time of actors. There is a cycle which consists of states and directed arrows, forming the periodic phase of the STE. The state transformations before the periodic phase constitute the transient phase, which features by imaginary lines in Figure 2.15b. A state in Figure 2.15b consists of two elements B and R. The three

elements in B show the current number of tokens on channels $A \rightarrow B$, $B \rightarrow A$ and $B \rightarrow B$, respectively. The two elements in R are the remaining execution time of invocations of actors A and B, respectively. Since the transient phase needs to execute only once, the IP of STE is decided by the period of its periodic phase. For the STE in Figure 2.15b, the period of the periodic phase is 1. Thus, the *IP* of the STE is 1.

The original STE approach is further extended to support other scheduling constraints. Two different buffer-constrained STE models are proposed in [SGB06a] and [ZGBS14], respectively. In [SGB06a], the buffer constraints are modelled as extra edges on the original SDFGs, while in [ZGBS14] the buffer constraints are modelled as extra variables in the states of the STE process. The effectiveness of the buffer-constrained STE is proven in [SGB06a]. Except for the constraint of buffer, the STE based scheduling is extended to support the constraint of a limited number of processing cores in [ZGBS12].

The advantages of STE based scheduling are listed below.

- If the number of cores is unlimited, the rate-optimal scheduling for strongly connected SDFGs is guaranteed by STE using the ASAP execution rule. This conclusion is confirmed by Chretienne in [Chr91] for HSDFGs and Ghamarian et al. in [GGS^{+}06] for SDFGs.

- The transient phase in the STE forms an effective retiming process which is always beneficial for getting better throughput and buffer size as shown in [ZGBS12] and [ZGBS14].

- Multiple iterations of an SDFG may exist in the periodic phase of STE, which equals to unfolding the original SDFG. The unfolding is also shown to be helpful for improving throughput and reducing buffer size in [ZGBS12] and [ZGBS14].

A disadvantage of STE based scheduling is that the time complexity of this kind of scheduling approaches is difficult to describe. In theory, in a worst-case scenario, the STE based scheduling has to traverse all possible states to find two identical ones. However, in practice, the runtime of STE based algorithms is always much faster than the worst cases. Another problem is that the number of states in an STE process may explode as the complexity of the scheduling problem increases, making the chance of finding two identical states to decrease dramatically. Thus, the periodic phase of STE becomes much harder to find for more complicated scheduling problem models. For example, if the communication overhead is considered in scheduling, extra

variables have to be added to states in the STE process to describe the occupation of communication resources. To investigate whether STE based scheduling approaches are still applicable for communication-aware scheduling problems, the original STE is extended in this thesis to support communication-aware scheduling problems. The details are given in Chapter 4.

2.8 Scheduling Algorithms

2.8.1 Scheduling Algorithms for Model $MD(X|F,N_s)$

The throughput of an SDFG on a single core system is always not affected by the scheduling of the SDFG [BL93]. Instead of the throughput, the main objectives of the scheduling on a single core are to minimize the code size and buffer size. For the inline based code generation approach, the work in [BL93] proposes **looped scheduling** which uses loops to compact the code blocks of repetitively executed actors. For example, for an SDFG containing three actors A, B, and C, a scheduling sequence $ABCBC$ can be expressed as $A(2BC)$, where the code blocks of actors B and C are embedded in a two-iteration loop. An algorithm called Pairwise Grouping of Adjacent Nodes (PGAN) is also proposed in [BL93] to find a schedule with the minimum code size. PGAN firstly constructs a clustered graph of the original SDFG by combining pairs of adjacent actors (or clusters) in the SDFG iteratively, until all actors have been combined into a hierarchically clustered graph. Then, the clustered graph is scheduled hierarchically using a rule to execute the multiple invocations of an actor sequentially. The sequential execution of an actor can be compacted in a loop and therefore the obtained code size is minimized.

The idea of using looped scheduling to reduce the code size of an SDFG is further developed in [BL94], where a well-known concept **single appearance schedule** (SAS) is proposed. A SAS only allows one appearance of each actor in the schedule sequence, which guarantees the minimum code size of the schedule. For example, the scheduling sequence $A(2BC)$ is a SAS, where actor blocks of A, B and C all appear only once. Besides constructing SASs to minimize the code size, in [BL94], a simple technique called *factoring schedule loops* is also proposed to reduce the buffer size of a SAS. In addition, a sufficient and necessary condition for the existence of a SAS for an SDFG is given in [BL94].

After SAS has been proposed, much work has been published based on SAS. A

scheduling framework is proposed in [BBHL95] which can achieve SAS to minimize code size based on the topological properties of an SDFG. The framework is flexible in that other scheduling objectives can also be jointly considered during the scheduling process, such as minimizing buffer requirements or increasing the number of data transfers that occur in registers. A fact is also mentioned in [BBHL95] that an acyclic SDFG (ASDFG) must have at least one SAS.

The work in [MBL97] proposes a dynamic programming (DP) algorithm to construct SASs while minimize buffer use for chain-structured SDFGs. In addition, the proposed DP algorithm is further extended in this paper for well-ordered SDFGs and ASDFGs with or without initial tokens on channels. The NP-completeness of constructing a SAS with minimum buffer size for ASDFGs is also proven in [MBL97]. Finally, a heuristic called RPMC is proposed to solve the buffer size minimization problem for SASs of ASDFGs.

An Evolutionary Algorithm (EA) based approach is proposed in [TZB98] and improved in [ZTB00], which aims to get a SAS with the minimum buffer size for an ASDFG. EA is used to explore the search space of actor execution orders. A DP based post-optimization step is used to get the minimum buffer use of an obtained actor execution order. According to the experiments in [TZB98] and [ZTB00], the runtime of the proposed EA based approach is longer than previous work but the performance is better.

The memory sharing of buffers on different channels is considered in [MB01] for getting SASs for ASDFGs. The buffer size can be efficiently decreased by buffer sharing compared to non-shared buffers in previous work. Several effective lifetime analysis algorithms for the data on buffers are proposed. All these analysis algorithms can be completed in polynomial time. A well-known first-fit heuristic is used to assign buffers on memory and get the required buffer size.

Another work considering shared buffers to reduce the total buffer size is the post-scheduling buffer management framework proposed in [FHHG12]. The details of the lifetime of the data on buffers are carefully analyzed. An ILP based algorithm and an EA based algorithm are proposed to solve the buffer minimization problem. The granularity of the optimization analysis can be changed by varying the analysis accuracy of the temporal behaviour of buffers. There is a trade-off between the runtime of the optimization and the granularity of the analysis.

An exact approach to get the minimum buffer for a SAS of an ASDFG is proposed in [KMB04]. The approach keeps decomposing an ASDFG according to a SAS until

achieving a graph of connected hierarchical actor pairs. As proven in [BML12], the minimum buffer size for a two-actor ASDFG is

$$B_{min} = p + q - gcd(p, q), \quad (2.11)$$

where p and q is the data producing rate and consuming rate of the two actors in the SDFG. Based on Equation 2.11, the minimum buffer size of the original SDFG can be recursively obtained from the graph of hierarchical actor pairs. An extension work for [KMB04] is proposed in [KMB07], where cyclic SDFGs are supported.

A buffer merge technique is proposed in [MB04], which enables the overlaying of buffers of a pair of input and output channels of an actor. Different from the buffer sharing strategy in [MB01], only the buffers of a pair of input and output channels of an actor can be shared. A conservative assumption for the timing behaviour of the lifetime of the data on buffers is that the data on the buffer of an input channel of an actor may not be released until the execution of the actor has finished and the buffer space on an output channel of an actor should be occupied by the output data at the beginning of the execution of that actor. By using a more detailed timing analysis of data releasing and generating on buffers, the buffer use is decreased in [MB04] compared with the conservative assumption. In [MB04], a mathematical buffer merging technique and several scheduling heuristics are proposed to reduce buffer size of SASs for ASDFGs.

Although SAS can provide minimum code size, not every SDFG can have a SAS according to [BL94]. References [MBL97, TZB98, ZTB00, MB01, KMB04, MB04] are all targeting to get SASs for ASDFGs as there is at least one SAS existing for an ASDFG [BBHL95]. There is also a trade-off between code size and buffer size as mentioned in previous work like [BBHL95]. To overcome these limitations of the SAS technique, some improvement work is proposed to minimize the code size of more generic schedules.

Instead of using the traditional SAS technique, the work in [SKH98] uses both inline based and function based approaches jointly to reduce code size and buffer size. A decision function is used to decide if an actor should be shared as a function. The overhead brought by function invocations is controlled by the decision function. However, an SAS is still needed as an input for this work. The method in [MBL97] is used to get the input SAS in this work.

The work in [Bjo04] proposes a code synthesis method for inline based code generation, which can avoid the duplication for the code blocks of actors for arbitrary schedules. Thus, SAS is not necessary for minimizing code size and buffer size can be

decreased without the constraint of SAS. The method constructs a finite state machine from a schedule. Then, the finite state machine is translated into an S-Graph with fewer states. Lastly, using loops and if-else statements, the S-Graph can be interpreted to the code where the code block of each actor only appears once.

A novel SAS technique called dynamic loop count SAS (dlcSAS) is proposed in [ODH06] to make an arbitrary schedule of an SDFG to be SAS. The loop counts of the loops in dlcSAS are not static, but change in a quasi-static way. The loop counts of loops are stored in arrays. For example, a schedule sequence *ABCA* cannot be converted to a SAS in a traditional way. However, a dlcSAS for the sequence is $2(A\{1,0\}(BC))$, where $\{1,0\}(BC)$ means the loop count of the loop containing B and C changes between 1 and 0 every time the loop has been invoked. Therefore, every schedule can have minimum code size on a single core using the dlcSAS technique. Thus, the scheduling of SDFGs can be focused on other optimization objectives instead of code size. Experimental results show that the proposed dlcSAS only sacrifices 1% performance overhead compared with the traditional SAS approach.

There is also some other work for specific single core systems. The work in [CC10] considers the accessing of off-chip memory and proposes a 3-stage ILP scheduling algorithm for a scratchpad memory (SPM) based processor, which minimizes code fetches from off-chip memory to reduce code overlay overhead under the constraint of memory size. An extended work based on [CC10] is proposed in [CC13], where a fast heuristic is also proposed except for the 3-stage ILP approach. In addition, the overlapping of code overlays and actor execution has been taken into consideration to reduce the code overlay overhead in [CC13]. The work in [LO13] proposes an answer set programming based approach for PRAM/DRAM hybrid systems to maximize the lifetime of data on PRAM, minimize DRAM use and minimize the energy of the systems by assigning data on appropriate memory types.

Summary

A summary of the scheduling algorithms for model $MD(X|F,N_s)$ is given in Table 2.1. The deadlock-free constraint for scheduling algorithms in Table 2.1 is omitted if other constraints exist. According to Table 2.1, for single core systems, most algorithms aim at minimizing buffer size under the constraint of SAS which guarantees the minimum code size. However, approaches proposed in [Bjo04] and [ODH06] can transfer any schedule to SAS, making the buffer size minimization problem to be independent from the constraint of code size.

Table 2.1: Scheduling algorithms for the model $MD(X|F,N_s)$.

Reference	Graph	Objective	Constraint	Accuracy
[BL93]	SDFG	code size	deadlock-free	heuristic
[BL94]	SDFG	buffer size	SAS	heuristic
[BBHL95]	SDFG	buffer size	SAS	heuristic
[MBL97]	ASDFG	buffer size	SAS	heuristic
[TZB98]	ASDFG	buffer size	SAS	heuristic
[MB01]	ASDFG	shared buffer size	SAS	heuristic
[FHHG12]	SDFG	shared buffer size	given schedule	exact & heuristic
[KMB04]	ASDFG	buffer size	SAS	exact
[KMB07]	SDFG	buffer size	SAS	heuristic
[MB04]	ASDFG	shared buffer size	SAS	heuristic
[SKH98]	ASDFG	code size & buffer size	deadlock-free	heuristic
[Bjo04]	SDFG	code size	deadlock-free	exact
[ODH06]	SDFG	code size	deadlock-free	exact
[CC10, CC13]	SDFG	code overlay overhead	on-chip memory size	heuristic
[LO13]	SDFG	lifetime of data on PRAM & DRAM use & energy	given schedule	exact

2.8.2 Scheduling Algorithms for Model $MD(X|F)$

For the model $MD(X|F)$, throughput and buffer size are both affected by scheduling. If throughput is the only objective, the model becomes $MD(TH|F)$. As mentioned in Section 2.5.5, the scheduling problem $MD(TH|F)$ for HSDFGs can be solved in polynomial time. In [Ram74], the scheduling problem $MD(TH|F)$ is transferred to a shortest-path problem and solved in polynomial time for HSDFGs. However, unfolding is not considered in [Ram74].

For generic SDFGs instead of HSDFGs, an LP based approach is proposed in [BMKdD12] for the problem model $MD(TH|F)$. In one iteration of a schedule for an SDFG, each invocation of an actor α can have one individual starting time. In [BMKdD12], to simplify the scheduling problem for SDFGs, the multiple invocations of an actor α_i within one iteration are restricted to have k_i individual periodic starting time. A K-vector is used to represent the number of individual starting time of actors in an SDFG. Each individual starting time of the invocations of an actor in one iteration is declared as a variable in the LP formulations. The complexity of the scheduling problem

increases with the size of the K-vector, which decides the number of inequalities to be solved by the LP solver. Therefore, the maximum throughput under different K-vectors is examined in [BMKdD12] to explore the trade-offs between the runtime of the solution algorithm and the maximum achievable throughput under the constraint of a K-vector.

When only buffer minimization is considered for the model $MD(X|F)$, the model becomes $MD(B|F)$, an NP-complete problem [BML12]. A heuristic is proposed in [ALP97] to get near-minimum buffer size for ASDFGs. Based on this, an extension work to process cyclic SDFGs is given in [Adé96]. An exact solution for buffer size minimization is proposed in [GBS05] based on model-checking. The model-checking approach is used to check if any deadlock-free schedule exists for an SDFG under the constraint of certain buffer size. As the lower bound and upper bound of the buffer size for an SDFG are easy to get, binary search can be used jointly with the model-checking approach to get the minimum buffer size that enables the existence of at least one deadlock-free schedule for the SDFG. There is also some work for specific memory structures. For example, in [RC01], a buffer binding algorithm is proposed to reduce the buffer size specifically for a memory structure using two-port buffers.

In most situations, throughput and memory use are both significant optimization objectives during scheduling. Therefore, much research is also performed to explore the trade-offs between throughput and buffer size or to minimize buffer size under a throughput constraint.

An STE based algorithm is proposed in [SGB06a] to explore the trade-offs between throughput and buffer size for SDFGs by solving the throughput-buffering Pareto points of SDFGs. During the exploration process, the buffer size on channels increases step by step. The constraint of the buffer size on a channel is modelled by adding a back-pressure edge for the channel to the original SDFG. The throughput under different distributions of the buffer size on channels is calculated by the STE based throughput analyzing approach introduced in [GGS+06]. A Pareto point is obtained by selecting the buffer distributions that have the minimum total buffer size under the same throughput. The effective accelerating approach is also proposed which sacrifices minor performance to gain significant speedups for the runtime of the exploration algorithm.

For the buffer size minimization problem under the rate-optimal constraint for HSDFGs, an LP based solution is proposed in [NG93], which claims to be able to get the exact minimum buffer size. However, this work has been disproved in [MBGS10]. The NP-completeness of the buffer size minimization problem under the rate-optimal constraint for HSDFGs has also been proven in [MBGS10]. An STE based algorithm

Table 2.2: Scheduling algorithms for the model $MD(X|F)$.

Reference	Graph	Objectives	Constraints	Retiming	Unfolding	Accuracy
[Ram74]	HSDFG	throughput	deadlock-free	Yes	No	exact
[BMKdD12]	SDFG	throughput	K-vector	Yes	No	exact
[ALP97]	ASDFG	buffer size	deadlock-free	No	No	heuristic
[Adé96]	SDFG	buffer size	deadlock-free	No	No	heuristic
[GBS05]	SDFG	buffer size	deadlock-free	Yes	Yes	exact
[RC01]	SDFG	buffer size	two-port buffers	No	No	heuristic
[SGB06a]	SDFG	buffer size & throughput	deadlock-free	Yes	Yes	near-exact
[MBGS10]	HSDFG	buffer size	rate-optimal	Yes	Yes	near-exact
[ZGBS14]	SDFG	buffer size	rate-optimal	Yes	Yes	heuristic

proposed in [SGB06a] is used in [MBGS10] as an exact solution for the buffer size minimization problem for rate-optimal scheduling.

A drawback of previous STE based work, like [SGB06a], is that initial tokens are viewed as real data, which affects the lower bound of the buffer size. Therefore, the minimum buffer size is usually unable to obtain for these work. In fact, initial tokens do not occupy any buffer space since they are only used to describe data dependencies between different iterations. Based on this fact, a Memory Constrained STE (MC-STE) is proposed in [ZGBS14] to eliminate the effect of initial tokens for buffer size. Using the proposed MC-STE and a binary search jointly, a heuristic to minimize buffer size for rate-optimal scheduling is proposed in [ZGBS14].

Summary

A summary of the scheduling algorithms for the model $MD(X|F)$ is shown in Table 2.2. For scheduling using unlimited number of processing cores, the minimum buffer size and maximum throughput can be obtained by existing work [BMKdD12] and [GBS05], respectively. However, the problem $MD(B|F,TH_{opt})$ has only near-optimal solutions in existing work [SGB06a, MBGS10, ZGBS14]. Therefore, the exact solution for $MD(B|F,TH_{opt})$ remains an open problem.

2.8.3 Scheduling Algorithms for Model $MD(X|F,N)$

Homogeneous Multicore Systems

If throughput is the only objective for scheduling, the scheduling model becomes $MD(TH|F,N)$, which has proven to be NP-complete for HSDFGs in [DGGH92]. A scheduling-range chart technique is also proposed for HSDFGs in [DGGH92], based on which a heuristic to maximize throughput under a fixed number of cores and a heuristic to minimize the number of cores under a throughput constraint are given. A scheduling range for an actor is the time interval where the actor can start its execution. By appropriately selecting the starting time of actors within their scheduling ranges, the objective of maximizing throughput or minimizing the number of cores can be achieved.

For the problem model $MD(X|F,N)$, an exact solution for HSDFGs is proposed in [BLMB09]. The solution is a CP based approach. A graph transformation based method is proposed to express the mapping of actors on cores and the execution order of actors on the same core by adding extra edges on the original HSDFG. The work is improved in [BBLM10], where the runtime of the algorithm is shortened by developing techniques to prune the solution space without compromising optimality. The CP based algorithm is further extended to process SDFGs in a binding based way in [BLMB13].

For SDFGs, a heuristic scheduling algorithm is proposed in [FKBS11] for throughput maximization. The heuristic tries to detect bottleneck actors. Then, a stream graph transformation process is used in the heuristic to execute the multiple invocations of a bottleneck actor in parallel by duplicating these actors on multiple cores if the bottleneck actor is possible to be duplicated. The limitation of memory size is considered in this work.

In [ZGBS12], an STE based scheduling algorithm is proposed to minimize the number of cores under the constraint of rate-optimal scheduling. The STE technique has been extended to support the constraint of the number of cores in this work. The throughput of an SDFG under the constraint of the number of cores is evaluated by the extended STE technique. A binary search based heuristic is proposed to minimize the number of cores for rate-optimal scheduling.

The work in [TBG$^+$17a] develops binding based mapping approaches to improve the throughput of SDFGs on multicore systems. A parallelism graph (PG) is proposed to illustrate the possibilities of parallel execution for actors. Then, the mapping problem for an SDFG can be transformed into a graph partition problem for the PG with a maximum cut value. An ILP based algorithm and a fast heuristic are proposed to solve

the graph partition problem. A hybrid genetic algorithm is also proposed based on the idea of improving the parallelism for a mapping.

Except for throughput, the buffer size may also be used as an objective during scheduling. For example, the work in [CZ12] optimizes buffer size under the constraint of throughput for ASDFGs. A two-level heuristic is proposed in this work for pipelined scheduling of ASDFGs. The inner level gets the optimal buffer size using a DP based approach for a list of actors in a given order, and the outer level tries to find the optimal order of the actors in terms of buffer size. However, the limitation of this paper is obvious. Firstly, only ASDFG is considered and multiple invocations of an actor are enforced to be consecutive, making the targeting SDFGs actually become DAGs. Secondly, only inter-processor buffers are considered, while the buffer size for communication between actors on the same processors is not optimized in this work.

The work in [DPNA13] investigates the effects of the timing of memory allocation for SDFGs on multicore systems. The memory allocation process can be performed in three stages: pre-scheduling stage, post-scheduling stage and after untimed scheduling. Here, untimed scheduling only decides the order of actor execution instead of the starting time of each actor. Trade-offs between memory size and scheduling flexibility are discussed and illustrated experimentally for allocating memory in different stages.

For the fault-tolerant design for multicore systems, a rescheduling framework is proposed in [LKOH13] to minimize the throughput degradation for processor failures under the constraint of latency. The work assumes a target multicore system may have at most one permanent processor failure. The proposed fault-tolerant algorithm is used to get reschedules of all the fault scenarios during compile time and applies the corresponding reschedule when a failure is detected during runtime. Preemptive and non-preemptive task migration policies for rescheduling process are discussed, based on which a hybrid policy is proposed.

Heterogeneous Multicore Systems

The heterogeneity of processing cores is also supported in the literature. The work in [ZPB^{+}11] proposes a framework for the design of Software-Defined Radio system on heterogeneous multiprocessors, where SDFGs are used to describe the target applications. The unfolding technique is used in this work. By changing the unfolding factor, the trade-offs between latency and throughput of a system are exploited. A MILP based scheduling algorithm is proposed for the maximization of throughput. However, this work only aims at ASDFGs and an SDFG-to-DAG transformation is

required for the scheduling process. An extension of this work is given in [ZPB+13], which applies the proposed framework on GPUs.

An ILP based scheduling algorithm is proposed in [LSGE11] to optimize the throughput, latency and processor cost jointly for SDFGs on heterogeneous multiprocessors. Here the processor cost can be a user-defined measurement for the area of processors, the price of processors, etc. The actor-to-core mapping is binding based. As the ILP based algorithm is time-consuming and not applicable for large SDFGs, a fast two-stage heuristic is also proposed, which consists of an EA based mapping stage and an ILP based scheduling stage. The EA is used to get a sub-optimal mapping for throughput and processor cost. Then, an ILP based scheduling algorithm minimizes the latency of the obtained mapping. By combining the two stages in different ways, the two-stage heuristic can generate 2-dimensional Pareto points for throughput and processor cost or 3-dimensional Pareto points for throughput, processor cost and latency.

In [HQR15], a throughput-constrained scheduling heuristic is proposed for applications in the area of Advanced Driver Assistance Systems, which are modelled as HSDFGs on a heterogeneous multicore system. The heuristic schedules the actors in an HSDFG one by one. During the scheduling process, the scheduling decision of an actor is expressed by adding extra actors and edges to the original HSDFG. The throughput of the half-scheduled HSDFG is obtained by calculating the MCM of the half-scheduled HSDFG. If the throughput of an HSDFG becomes lower than the throughput constraint after an actor has been scheduled on a processor, then the actor is rescheduled onto another processor. If the throughput constraint cannot be satisfied by scheduling the actor on any of the processors, a rollback process should occur which reschedules the previously scheduled actor.

A scheduling workflow is proposed in [ZSJ09] for the CPU/FPGA hybrid multicore architecture. The mapping of actors to cores is binding based and assumed to be given in advance of the scheduling. The buffer size is minimized under the constraint of throughput. A CP based approach is used to solve the scheduling problem. Although this work claims to aim at a CPU/FPGA hybrid architecture, the scheduling problem solved in this work is independent from the heterogeneous architecture since the actor-to-core mapping is given.

Energy consumption is another widely used optimization objective for heterogeneous multicore systems except for throughput. In [AHSvdP15], a model-checking based scheduling algorithm is proposed to get optimal energy

consumption under the throughput constraint for heterogeneous multiprocessors using both Dynamic Power Management (DPM) and Dynamic Voltage and Frequency Scaling (DVFS) techniques. An extension work for [AHSvdP15] is given in [AJSvdP16], where the model-checking approach is also used to optimize the lifetime of batteries of a system or the battery capacities used to complete a workload under a throughput constraint. The use of multiple batteries is supported. For power reducing techniques, except for DPM and DVFS, the voltage-frequency islands (VFIs) technique is also considered in this work.

Another model-checking based scheduling algorithm is proposed in [ZYG⁺15] to get optimal throughput with the best energy consumption or optimal energy with the best throughput under the constraints of auto-concurrency and buffer size for heterogeneous multiprocessors. Unfolding is considered in this work, yet the unfolding factor is assumed to be given as an input of the algorithm.

The work in [GZZ15] proposes a design space exploration approach to get Pareto points between throughput and energy consumption for SDFGs on heterogeneous multicore systems. The unfolding technique is used in this work. An extended work for [GZZ15] is proposed in [AvdP16], which provides an exhaustive approach and a heuristic algorithm to explore the trade-offs between throughput and energy consumption.

For the quality of systems (QoS), the work in [MAG12] proposes a process-variation aware mapping algorithm for SDFGs on heterogeneous MPSoCs to improve yield. Here, the process variation means the maximum supported frequency variation for processing cores on a chip during the manufactory process; and the yield means the number of manufactured chips that satisfy the application timing requirement.

Summary

A summary of scheduling algorithms for the model $MD(X|F,N)$ is given in Table 2.3, where the column heterogeneity shows the heterogeneity of processing cores supported by algorithms. The constraint of the number of cores for scheduling is omitted in Table 2.3 as this constraint is applied to all algorithms in this section except that the number of cores is used as an optimization objective for an algorithm.

As the number of processing cores is considered in this model, the complexity of solving scheduling problems for this model is harder than the single core model $MD(X|F,N_s)$ or the unlimited core model $MD(X|F)$. Therefore, in order to simplify the scheduling problem, some work only aims at HSDFGs [DGGH92] or needs to convert

Table 2.3: Scheduling algorithms for the model $MD(X|F,N)$.

Reference	Graph	Objectives	Constraints	Retiming	Unfolding	Accuracy	heterogeneity
[DGGH92]	HSDFG	throughput / number of cores	deadlock-free	Yes	No	heuristic	homo
[BLMB09, BLMB13]	HSDFG	throughput	deadlock-free	Yes	Yes	exact	homo
[BLMB13]	SDFG	throughput	binding based mapping	Yes	Yes	exact	homo
[FKBS11]	SDFG	throughput	memory size	No	No	heuristic	homo
[ZGBS12]	SDFG	number of cores	rate-optimal	Yes	Yes	heuristic	homo
[TBG+17a]	SDFG	throughput	binding based mapping	No	No	heuristic	homo
[CZ12]	DAG	buffer size	throughput	No	No	heuristic	homo
[DPNA13]	SDFG	memory & scheduling flexibility	deadlock-free	No	No	heuristic	homo
[LKOH13]	SDFG	throughput	latency of rescheduling	No	No	heuristic	homo
[ZPB+11, ZPB+13]	DAG	throughput	unfolding factor	Yes	Yes	exact	hetero
[LSGE11]	SDFG	throughput & latency & processor cost	binding based mapping	Yes	No	heuristic	hetero
[HQR15]	HSDFG	deadlock-free	throughput	Yes	Yes	heuristic	hetero
[ZSJ09]	SDFG	buffer size	throughput & given binding based mapping	Yes	Yes	exact	hetero
[AHSvdP15]	SDFG	energy	throughput	Yes	No	exact	hetero
[AJSvdP16]	SDFG	lifetime of batteries / battery capacities	throughput	Yes	No	exact	hetero
[ZYG+15]	SDFG	energy / throughput	throughput/energy & buffer size	No	Yes	exact	hetero
[GZZ15, AvdP16]	SDFG	throughput & energy	unfolding factor	Yes	Yes	exact & heuristic	hetero
[MAG12]	SDFG	yield	throughput & binding based mapping	Yes	Yes	exact & heuristic	hetero

an input SDFG to an equivalent HSDFG [BLMB09, BLMB13, HQR15] or further convert to a DAG [CZ12, ZPB$^+$11, ZPB$^+$13] before scheduling. Besides, binding based mapping is also widely used as a simplification approach, e.g. in [BLMB13, LSGE11, ZSJ09, MAG12]. Objectives such as throughput, buffer size, latency, energy, etc. are well optimized through exact or heuristic scheduling algorithms in existing work.

Although restricting the actor-to-core mapping to be binding based can simplify the scheduling problem, the throughput of a multicore system is degraded as well since the data-level parallelism within an actor cannot be utilized. Therefore, duplication-enabled mapping is preferable for high-throughput demand applications. However, the duplication of the code blocks of actors on multiple cores could lead to an increase of the overall code size of an application. In addition, if the unfolding technique is also used, the possibility for an actor to be mapped on multiple cores is further increased as the number of invocations for an actor is multiplied by the unfolding. Therefore, for the scheduling using duplication-enabled mapping, an open problem is to reduce the overall code size of an SDFG on a multicore system during scheduling.

2.8.4 Scheduling Algorithms for Model $MD(X|F,N,C)$

For the model $MD(X|F,N,C)$, communication overhead has been taken into consideration. The communication architectures can be categorized roughly into four: fully connected, bus-based, NoC based, and heterogeneous. For fully connected systems, processing cores communicate with each other through dedicated channels without communication contention. Bus-based architectures use one or multiple shared buses to connect all processing cores in a system. As only a limited number of simultaneous inter-core communication can be supported by a bus, communication contention may happen on bus-based systems. Therefore, except for computing tasks, the scheduling for bus-based systems should also handle the communication tasks properly to avoid communication contention. For a NoC based system, processing cores are located on distributed tiles connected by an on-chip network. Each tile contains a router responsible for sending, receiving and forwarding of data packets. There are different routing strategies which decide the path for the communication between two tiles. As for systems adopting heterogeneous communication architectures, a hybrid of multiple different communication mechanisms (e.g. shared memory, on-chip network, etc.) may exist in these systems. Scheduling for these systems also includes deciding the communication mechanism used for each

communication task.

Fully-Connected Systems

For fully-connected systems, the work in [ZDP+13] proposes a two-phase scheduling framework to optimize throughput for a Fully Parallelizable Parallel Actor Scheduling (FP-PAS) problem for SDFGs on homogeneous multicore systems. A parallel actor has inherent parallelism which can be accelerated by executing the actor on multiple processing cores. If every actor in an SDFG is a parallel actor, the scheduling for this SDFG is called FP-PAS. For the two-phase scheduling framework, an SDFG is transformed to a DAG before scheduling. Then, in phase one, a particle swarm optimization engine is used to generate processor assignments for actors. After that, in phase two, a MILP based approach or a fast list scheduling heuristic is used to solve the assignment-constrained FP-PAS problem. The shared memory based communication architecture used in the work is equivalent to a fully connected architecture except that the memory writing of the parallel execution of the same actor must be in sequence.

A SAT-based scheduling approach is proposed in [LGY15] for HSDFGs on homogeneous multiprocessors. Actor retiming is considered in this work. A SAT solver is used to explore the design space of actor retiming, mapping and ordering. One candidate solution of retiming, mapping, and ordering for an HSDFG is modelled by adding extra actors and edges to the original HSDFG. Then, the throughput of the candidate solution is analyzed through the transformed HSDFG. A branch-and-bound (BNB) process is used jointly with the SAT solver to get the maximum throughput. Communication overhead is modelled as a constant delay for each token without communication contention. Therefore, the considered communication architecture is equivalent with a fully-connected architecture.

Bus-based Systems

For bus-based systems, in [KBB06], the ordering for communication tasks is shown to be NP-complete for maximizing the throughput of a scheduled HSDFG where the mapping of actors and the ordering of actors are given. Communication tasks are assumed to occupy a processor during the data sending or receiving process. The bus used in [KBB06] supports the inter-processor communication of at most one pair of processors and denies other communication requests if the bus is occupied. An IPC graph is constructed from a scheduled HSDFG. Based on the IPC graph, three heuristics are proposed, including a transaction partial order (TPO) heuristic, a genetic algorithm

(GA) based approach, and a sifting algorithm [Rud93] based dynamic transaction reordering (DTR) approach. A BNB based algorithm is also given as an exact solution. In addition, the three proposed heuristics TPO, GA and DTR can be combined together to get near-optimal results according to the experiment.

An extension work for [ZDP+13] is given in [ZPB+16], where SDFGs with non-parallel actors are supported and a MILP based approach is proposed to solve this problem. In addition, a new simulation annealing based heuristic is proposed in [ZPB+16] for the phase two of the two phase framework proposed in [ZDP+13] for the FP-PAS problem. The target communication architecture in [ZDP+13] has changed from the fully-connected architecture in [ZDP+13] to a more practical bus-based architecture.

A GA based scheduling algorithm is proposed in [COKH12] to minimize the overhead of memory overlay for SDFGs on SPM-based heterogeneous multicore systems. The target SPM-based multicore system consists of core tiles and memory tiles connected with a bus. Each core tile consists of a processing core, an SPM-based local memory and a DMA unit. The overhead of code overlay and data overlay on local memory is minimized to reduce latency by using aggressive prefetching. The binding based mapping strategy is used in this work.

The work [LSGE11] is extended in [LGE12] by considering communication overhead. The target communication architecture consists of heterogeneous cores with local memory and a shared memory connected by multiple buses. There are two routing policies for the interprocessor communication: directly communicating through a bus and two-hop communication via the shared memory. The data sending and receiving time is modelled as additional communication actors in the original SDFG. The communication policy selection for each interprocessor communication and the bus assignment for the communication actors are both solved during scheduling. The global ILP scheduling algorithm and the two-stage heuristics proposed in [LSGE11] are extended in [LGE12] to support the communication-aware scheduling.

A CP based scheduling framework is proposed in [RS17] for scheduling streaming applications on heterogeneous multicore systems. The framework supports the scheduling of multiple applications running on a system with various scheduling constraints, e.g. buffer size, throughput, power, etc. A CP solver is used with a BNB process to optimize multiple objectives like throughput, power, and buffer size. The streaming applications modelled by SDFGs have to be converted to HSDFGs before scheduling. An HSDFG transformation based modelling technique is used to model

scheduling results on the original HSDFG. Throughput analysis techniques are used on the transformed HSDFGs to get the throughput of schedules. In this work, interprocessor communication is through a bus using a time-division multiple-access (TDMA) technique, which gives each processor a dedicated communication time slot to prevent communication contention.

NoC-based Systems

For NoC based systems, a scheduling heuristic is proposed in [SBGC07] to achieve high-efficiency hardware resource allocations for multiple applications modelled by SDFGs on a heterogeneous multicore system to meet throughput constraints of these applications. Binding based mapping is used in this work. The concurrent execution of multiple applications on a processor works in a TDMA way. The actor to core allocation is instructed by a cost function. Except for the throughput constraint, the constraints of memory, communication bandwidth and time slices on processors are also considered during the resource allocation process.

The work in [ZSJ10] proposes a CP based scheduling algorithm to construct efficient schedules for actors and contention-free routing for data communication on NoC based multicore systems for SDFGs. The actor-to-core mapping is binding based and assumed to be given in advance. The objective of the proposed scheduling algorithm is to minimize buffer size under the constraint of throughput.

A scheduling heuristic is proposed in [HFG13] to explore the trade-offs between memory size and throughput for SASs of ASDFGs on NoC-based multiprocessors. The overlapping of different iterations of an SDFG is enabled under the constraint of memory size in this work. The FIFO-based data communication on channels of an SDFG is guaranteed for NoC-based architectures in this work.

For the fault-tolerant design for NoC-based multimedia multiprocessors, an ILP based remapping algorithm is proposed in [DKV12] to minimize the energy consumption of a remapping process. As claimed in this work, the energy consumption for communication accounts for more than 40% of the overall energy consumption for most of multimedia systems. Thus, the work focuses on reducing communication energy. For each fault scenario, a new mapping with minimum communication energy consumption under a throughput constraint can be obtained by existing approaches. The proposed remapping algorithm is used to minimize the energy overhead of the task migration during remapping from the original mapping to the new mapping. A more complete extension of this work is given in [DKV14], where heuristics are proposed for

throughput constrained SDFGs on heterogeneous multiprocessors to generate energy-aware mapping, fault-tolerant remapping and self-timed scheduling. SDF3 [SGB06b] is used in this work to evaluate the throughput and energy consumption of a mapping. The remapping in [DKV12, DKV14] is binding based.

A model-checking inspired scheduling approach is proposed in [MG15] to optimize energy consumption and throughput simultaneously for NoC based heterogeneous multicore systems. The scheduling problem is modelled as a reachability tree based on one of the model-checking languages named Computational Tree Logic formulas [BCM+92]. Then, breadth-first search for the reachability tree is used to solve the scheduling problem. Effective path-pruning techniques are also used in the search, including branch and prune, divide and conquer and relaxed greedy allocation.

In [DKV16], a scheduling approach is proposed to improve the lifetime reliability and minimize energy consumption for throughput constrained SDFGs on multiprocessors supporting multiple voltage-frequency pairs. Firstly, a simplified temperature model is proposed in this work. Based on this temperature model, a fast gradient-based heuristic is proposed to set the voltage and frequency of cores, which optimizes the peak and average temperature of cores, resulting in longer lifetime and lower energy consumption. Remapping is adopted as a fault-tolerant technique to prolong the lifetime of a system. To support concurrent execution of multiple SDFGs on a multiprocessor, an LP based algorithm is proposed to allocate processing cores for concurrent applications while optimizing the lifetime of the system. Binding based mapping is used in this work.

The work in [TBG+17b] proposes an iteration-based task-FIFO co-scheduling (ITFCS) framework for multicore systems based on NoC using TDM scheme [SMG14]. Binding based mapping is used in this work. The architecture of memory consists of two hierarchies: the local memory on tiles with limited size and a large enough off-chip memory. Different accessing latency of on-chip memory and off-chip memory is considered. In terms of the scheduling process, firstly, the proposed ITFCS framework uses the design space exploration algorithm proposed in [SGB06a] to get the Pareto points of buffer size and throughput. Then, a buffer-allocation-aware mapping algorithm is proposed to allocate actors to processors and buffers to on-chip or off-chip memory while minimizing the overall memory accessing cost. The buffer allocation and memory accessing time are modelled by adding extra actors and edges to the original SDFG. Finally, a memory-aware earliest task first scheduling (MAETF) algorithm which allows task insertion is used to evaluate the throughput of the buffer allocation and actor-to-core

Table 2.4: Scheduling algorithms for the model $MD(X|F,N,C)$.

Reference	Graph	Communication	Objectives	Constraints	Retiming	Unfolding	Accuracy	heterogeneity
[ZDP+13]	DAG	fully-connected	throughput	FP-PAS	No	No	heuristic	homo
[LGY15]	HSDFG	fully-connected	throughput	deadlock-free	Yes	Yes	exact	homo
[KBB06]	HSDFG	bus-based	throughput	given schedule	—	—	heuristic	—
[ZPB+16]	DAG	bus-based	throughput	PAS	N	N	exact & heuristic	homo
[COKH12]	SDFG	bus-based	memory overlay overhead	on-chip memory size & binding based mapping	No	No	heuristic	hetero
[LGE12]	SDFG	bus-based	throughput & latency & processor cost	binding based mapping	Yes	No	heuristic	hetero
[RS17]	HSDFG	bus-based	buffer size / throughput / energy	buffer size / throughput / energy	Yes	Yes	exact	hetero
[SBGC07]	SDFG	NoC based	resource efficiency	throughput & hardware resources & binding based mapping	Yes	Yes	heuristic	hetero
[ZSJ10]	SDFG	NoC based	buffer size	throughput & given binding based mapping	Yes	No	exact	—
[HFG13]	ASDFG	NoC based	throughput & memory	SAS & memory size	Yes	Yes	exact	homo
[DKV12, DKV14]	SDFG	NoC based	energy	throughput of remapping & binding based mapping	Yes	Yes	exact & heuristic	hetero
[MG15]	SDFG	NoC based	energy & throughput	deadlock-free	Yes	Yes	exact	hetero
[DKV16]	SDFG	NoC based	lifetime & energy	throughput & binding based mapping	Yes	Yes	heuristic	hetero
[TBG+17b]	SDFG	NoC based	throughput	on-chip memory size & binding based mapping	Yes	Yes	heuristic	homo
[MG13]	SDFG	heterogeneous	throughput	deadlock-free	Yes	Yes	exact	hetero

mapping and to get a schedule based on the mapping.

Heterogeneous Communication Architecture based Systems

For heterogeneous communication architectures, a model-checking based approach is proposed in [MG13] to schedule SDFGs on heterogeneous multiprocessors with heterogeneous communication architectures. The objective of the proposed scheduling approach is to maximize the throughput of a system. Task-level parallelism and data-level parallelism are both utilized by the proposed approach. A state sharing technique is also used in this work to improve the performance of a system.

Summary

A summary of scheduling algorithms for the model $MD(X|F,N,C)$ is given in Table 2.4. As communication overhead is considered in this model, the complexity of solving the scheduling problems for this model becomes extremely hard. Therefore, simplification

approaches like SDFG-to-HSDFG conversion, SDFG-to-DAG conversion or binding based mapping are used in almost all existing work for the model $MD(X|F,N,C)$. For throughput optimization, only the work in [MG13] gives an exact solution directly for SDFGs without simplification. However, exact solutions are time-consuming and not applicable for large SDFGs. Therefore, fast and efficient scheduling heuristics are also needed for the throughput maximization problem for the model $MD(X|F,N,C)$.

2.9 Evaluation Tools and Benchmarks for SDFGs

The performance of SDFGs on multicore systems can be evaluated by analyzers [SGB06b], software simulators [KMT+06] or FPGA simulators [KFH+08]. An analyzer evaluates the performance of a system by analyzing the behaviours of the system, while a software simulator or a FPGA simulator actually simulates the execution of an application on a multicore system through software simulation or FPGA simulation. Therefore, in general, the speed of analyzers is faster than the speed of simulators; and the speed of FPGA simulators is always faster than the speed of software simulators. On the other hand, the accuracy of analyzers is usually lower than the accuracy of simulators. The evaluation of FPGA simulators is always more accurate than software simulators.

In this thesis, a widely used open source tool [DKV14, CZ12, ZGBS12, ZGBS16, ZGBS14, BLMB13, TBG+17a, TBG+17b] named SDF3 [SGB06b] is adopted as an analyzer to evaluate the performance of SDFGs on multicore systems. In addition to performance analysis, SDF3 also provides a series of transformation, generation and scheduling functions for SDFGs. The algorithms in the scheduling framework proposed in this thesis are implemented in SDF3 as well. The website of SDF3 [SGB06b] provides benchmarks which involve realistic multimedia and signal processing applications like a H.263 encoder and a digital filter. Datasets which consist of SDFGs generated randomly by SDF3 are also used in the experiments in this thesis.

Another source of benchmarks is StreamIt [TKA02, Str18]. StreamIt is an open source compilation infrastructure and a programming language used to develop streaming applications. A series of realistic benchmarks implemented by the StreamIt language are provided in StreamIt, including an audio beam former, a fast Fourier transform (FFT) kernel, etc. The program analysis functions and the SDFG generation function in StreamIt can be used to construct SDFGs from the StreamIt code of the benchmarks and get necessary information for SDFG scheduling such as the workload

of each actor, the code size of each actor, the token size on each channel, etc.

2.10 Summary

This chapter defines the concept of the SDFG model used in this thesis and introduces some significant properties of SDFGs. After that, the advantages and limitations of two widely used SDFG transformation techniques, retiming and unfolding, are discussed. Then, scheduling problem models with various constraints and objectives are described, followed by the introduction of some modelling techniques of the constraints or scheduling results. Then, commonly used scheduling methodologies for SDFGs are discussed and a review of related literature is given. The development and open problems for SDFG scheduling are summarized based on the review of the related literature. Lastly, the evaluation approaches and benchmarks are introduced.

In this thesis, non-preemptive static scheduling is adopted as the scheduling strategy to be optimized. Throughput and memory usage are two optimization objectives of the scheduling framework proposed in this thesis. The STE based scheduling methodology is used to solve the scheduling problem for SDFGs on both homogeneous and heterogeneous multicore systems. The scheduling framework proposed in this thesis is implemented and evaluated on SDF3. Benchmarks from SDF3 and StreamIt are used in the experiments, as well as some random SDFGs generated by SDF3.

Chapter 3

Buffer Minimization for Rate-Optimal Scheduling

3.1 Overview

For streaming applications represented by SDFGs, one of the key scheduling objectives is throughput. A *rate-optimal* schedule is a schedule that can achieve the maximal throughput for a given SDFG. Under the constraint of the rate-optimal schedule, memory usage is a commonly considered optimization objective. Communication data between actors is modelled as *tokens*, where each token represents an application-specific set of data. The memory used to store the tokens exchanged between actors is referred to as *buffers*. The buffer size is represented by the number of tokens it can hold. This chapter addresses the problem of minimizing the buffer size for rate-optimal SDFG schedules on multicore systems.

On single-core systems, throughput is hardly affected by scheduling. Therefore, for these systems, research is focused on minimizing buffer requirements of SDFGs during scheduling [BL94, BBHL95, MBL97, TZB98, KMB04, KMB07]. However, for multicore systems which allow actors to fire in parallel on different cores, throughput and buffer usage are both influenced by the scheduling process. For example, for the SDFG shown in Figure 3.1, actor *A* fires twice and actor *B* and *C* fire once within one iteration of the execution. The execution time of actor *A*, *B* and *C* is 1 time unit, 1 time unit and 3 time unit, respectively. If all actors are scheduled onto one core, the iteration period (IP) of the graph should be the sum of the execution time of these firings, which is 6 time units. However, by firing actor *B* two times on two cores, actor *C* two times on two cores and actor *A* four times on 4 cores sequentially, two iterations can be

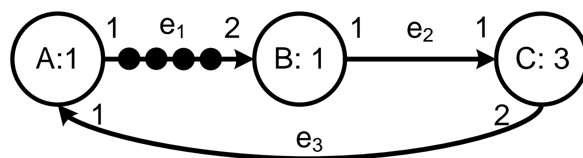


Figure 3.1: An example SDFG.

completed during a period of 5 time units, which means the period of one iteration is only 2.5 time units. Therefore, different schedules decide the throughput of an SDFG on a multicore system. Besides, the scheduling process is also constrained by the buffer sizes on channels. The firings of actors can be blocked by insufficient buffer space, something that leads to a further decrease of the throughput. Therefore, minimizing buffer usage under throughput constraints is a problem for multicore systems. For example, minimization of buffer usage is desirable in multicore embedded systems, since the on-chip memory on embedded systems is expensive and limited in size while the accessing time and energy consumption of off-chip memory is significantly larger than on-chip memory.

For the problem of minimizing the buffer size for rate-optimal SDFG schedules on multicore systems, existing algorithms all have their limitations: either they aim at a subset of SDFGs [MBGS10] or they cannot guarantee an optimal solution [SGB06a, ZGBS14, CZ12, SOH11, RS14]. One of the best heuristics is proposed by Zhu et al. in [ZGBS14], which can get near-optimal buffer usage for most SDFGs. However, as shown in Section 3.5, their solutions may still require as much as 50% more buffer usage than the optimal solutions for some extreme cases. This gives scope for an exact algorithm and/or a more effective heuristic.

The contributions of this chapter are the following:

- An extended self-timed execution method to eliminate the extra buffer requirements caused by initial tokens;
- An exact algorithm to find a rate-optimal schedule with minimal buffer requirements for an SDFG based on the searching algorithm in [SGB06a]. The execution time of the searching is decreased significantly by using more accurate bounds for searching than the bounds used in [ZGBS14];
- A fast and efficient heuristic for buffer usage minimization, which can be used

when the runtime of the scheduling process is constrained. The heuristic is based on the proposed exact algorithm and adopts the idea of the quantization factor in [SGB06a].

The remainder of the chapter is organized as follows. Section 3.2 explains the concepts used in this chapter. After that, a motivational example is given in Section 3.3. The proposed algorithms, including the exact and heuristic searching algorithms are introduced in Section 3.4. Then, experimental results are presented in Section 3.5. Finally, Section 3.6 concludes the chapter.

3.2 Background

3.2.1 Problem Definition

The objective of this chapter is to minimize buffer requirements for rate-optimal schedules of SDFGs on multicore systems. The input SDFGs are assumed to be sample-rate-consistent and deadlock-free. The data buffering in a multicore system can be realized on a shared memory space or on separate (distributed) memory spaces. In this chapter, a conservative buffer sharing assumption is used to guarantee the proposed algorithm can be applied on systems with various buffer sharing policies. The assumptions are listed below.

- The buffer on different channels cannot be shared with each other and unoccupied buffer space on a channel cannot be used by other channels;
- For a channel e, the buffer space is occupied in the beginning of the execution of its source actor $\alpha_{src}(e)$;
- For a channel e, the buffer space is released at the end of the execution of its destination actor $\alpha_{dst}(e)$.

The definition of the concepts of *buffer distribution* and *distribution size* are given in Definition 15.

Definition 15. A buffer distribution d is a mapping set for the channel set E of an SDFG $G = (V, E)$, where an element $d(e)$ corresponds to the buffer size on the channel e, $e \in E$. The distribution size is denoted by $|d| = \sum_{e \in E} d(e)$.

Then, the objective of this chapter can be stated as:

- **Find a schedule with the minimum $|d|$ which satisfies the condition $IP = LB$.**

The buffer size on a channel is denoted as the number of tokens the buffer can hold. As commonly observed from previous work [ZGBS14, SGB06a, MBGS10], the size of tokens on different channels is assumed to be identical. For SDFGs having various token size on channels, it is easy to normalize the token sizes of channels by letting $p(e) := p(e) * TS(e)$ and $q(e) := q(e) * TS(e)$ ($\forall e \in E$).

3.2.2 Buffer-Constrained Self-Timed Execution

Most analysis and scheduling methods are based on HSDFGs, so an SDFG-to-HSDFG conversion has to be used [SBGC07]. However, this conversion may cause an exponential increase in the number of actors of the converted HSDFG. To avoid this conversion, STE based approaches are proposed for the analysis or scheduling of SDFGs [GGS+06, SGB06a, SBGC07, ZGBS14, ZGBS12].

Based on the original STE technique, two different buffer-constrained STE (BC-STE) models are proposed in [SGB06a] and [ZGBS12], respectively. In [SGB06a], the buffer constraints are modelled as extra edges on the original graphs, while in [ZGBS14] the buffer constraints are mapped on real buffers and checked during the execution process. As discussed in Section 2.6.1, the modelling approach proposed in [SGB06a] cannot guarantee the achievement of the minimum buffer size since the influence of initial tokens is hard to be eliminated by this modelling approach. Therefore, the modelling approach proposed in [ZGBS14] is used in this thesis and extended to remove the effect of initial tokens. The definition of a state in BC-STE proposed in [ZGBS14] is given in Definition 16.

Definition 16. A state in a BC-STE is represented by three vectors $(\boldsymbol{T}, \boldsymbol{S}, \boldsymbol{R})$. $\boldsymbol{T}$ and $\boldsymbol{S}$ are both M-dimensional vectors, where M is the number of channels in an SDFG. Each element in $\boldsymbol{T}$ represents the amount of the tokens on a channel, and each element in $\boldsymbol{S}$ represents the remaining buffer space on a channel. The initial value of $\boldsymbol{T}(e)$ equals to $it(e)$. The initial value of $\boldsymbol{S}(e)$ is obtained by $d(e) - it(e)$. $\boldsymbol{R}$ is a N-dimensional vector, where N is the number of actors. Each element in $\boldsymbol{R}$ denotes the remaining execution time of an actor. Since multiple invocations of an actor may execute in parallel on different processing cores, each element of $\boldsymbol{R}$ is a multiset. At the beginning of the BC-STE process, the initial values of all elements in R are all empty multisets denoted by $\{\}$.

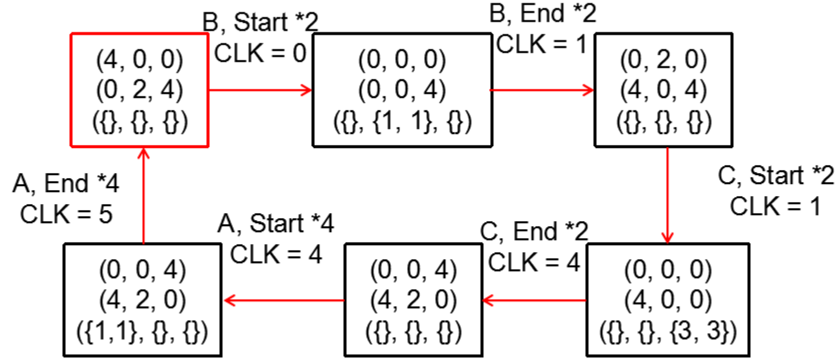


Figure 3.2: The BC-STE of the example SDFG in Figure 3.1.

Compared with Definition 14, as shown in Definition 16, an extra vector $\boldsymbol{S}$ is added to the state of BC-STE to represent the size of the unoccupied buffer space on channels. Except for the state of BC-STE, the state exploration process of BC-STE also takes the constraint of buffer size into consideration. The BC-STE process improves Algorithm 3 by updating three operation functions $end(\alpha, CurState)$, $startReady(\alpha, CurState)$ and $start(\alpha, CurState)$ in the algorithm to support the constraint of buffer size. The three buffer-constrained functions $end_{BC}(\alpha, CurState)$, $startReady_{BC}(\alpha, CurState)$ and $start_{BC}(\alpha, CurState)$ are defined in Equations 3.1, 3.2 and 3.3, respectively.

$$\begin{aligned} end_{BC}(\alpha, CurState) \equiv_{def} & \left(\forall e \in INPUT(\alpha), CurState.\boldsymbol{S}(e) := CurState.\boldsymbol{S}(e) + q(e)\right), \\ & \wedge \left(end(\alpha, CurState)\right). \end{aligned} \tag{3.1}$$

$$startReady_{BC}(\alpha, CurState) := \begin{cases} True, & if \ \left(\forall e \in OUTPUT(\alpha), CurState.\boldsymbol{S}(e) \geq p(e)\right) \\ & \quad \wedge \left(startReady(\alpha, CurState)\right); \\ False, & otherwise. \end{cases} \tag{3.2}$$

$$\begin{aligned} start_{BC}(\alpha, CurState) \equiv_{def} & \left(\forall e \in OUTPUT(\alpha), CurState.\boldsymbol{S}(e) := CurState.\boldsymbol{S}(e) - p(e)\right) \\ & \wedge \left(start(\alpha, CurState)\right). \end{aligned} \tag{3.3}$$

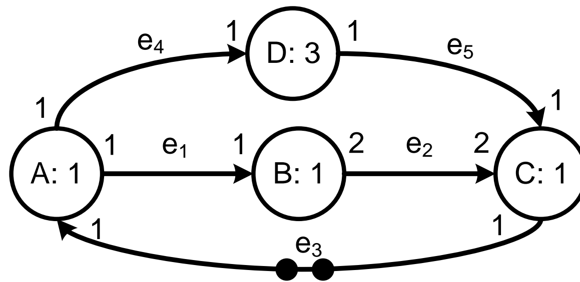
As an example, Figure 3.2 gives the BC-STE of the SDFG in Figure 3.1 according to the technique in [ZGBS14]. In Figure 3.2, the rectangles represent the states of the execution of the SDFG in Figure 3.1. The rectangle on the top left corner is the initial

state. The states are transformed by the actions shown between two states. *CLK* is the abstracted timing reference which labels the time when each action takes place. The time unit of *CLK* is the same as the time unit of the execution time of actors. There is a cycle which consists of states and directed arrows, forming the periodic phase of the BC-STE. A state in Figure 3.2 consists of three elements: the first row is the vector of the token number on channels ($\boldsymbol{T}$), the second row is the vector of the unused buffer sizes on channels ($\boldsymbol{S}$), and the third row is the vector of the remaining execution time of the firing actors ($\boldsymbol{R}$). No transient phase exists in this BC-STE process. The *IP* of this BC-STE is 5 time units. Since two iterations exist in this BC-STE, the average IP of one iteration is 2.5 time units, which equals to the LB of the SDFG in Figure 3.1. Therefore, this BC-STE forms a rate-optimal schedule where the minimum buffer requirements on channels e_1, e_2 and e_3 is 4, 2 and 4, respectively.

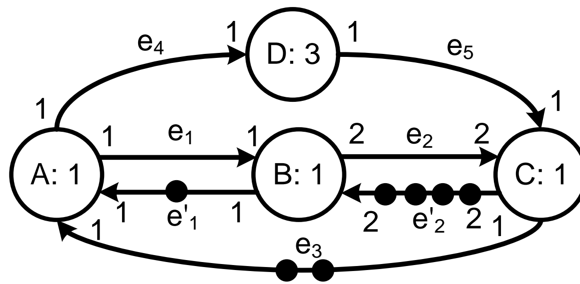
3.3 Motivational Example

Decreasing buffer requirements for systems modelled by SDFGs is a well-studied topic. Most of the existing work only aims at single-core systems. However, as the complexity of modern systems is getting higher, multicore architectures are more and more widely used and there is a lot of research on buffer minimization or throughput improvement [MBGS10, SGB06a, GBS05, ZGBS12, ZGBS14, SBGC07, CZ12, SOH11, RS14]. However, some research only pursues the minimization of the buffer requirements giving an executable schedule without considering the optimization of the throughput, e.g. the work in [BML12, GBS05]. Since throughput is always one of the most significant metrics of a streaming application, these papers may not be applicable for the design of throughput-intense systems. There is recently also some work that only considers throughput but no memory constraints in terms of buffer usage [DDLR15, DDL15, LGY15].

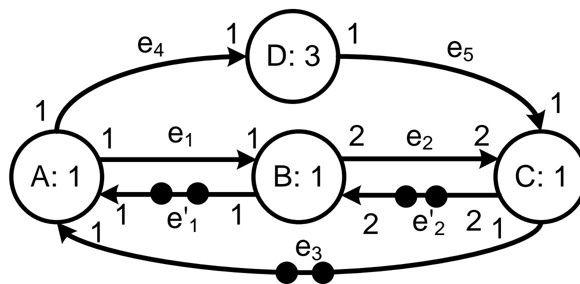
In [MBGS10], an algorithm is proposed to solve the buffer minimization problem for homogeneous SDFGs (HSDFGs), a subset of SDFGs. Since SDFGs can be converted to their equivalent HSDFGs, this algorithm can also be applied to SDFGs. However, the minimal buffer sizes obtained from the HSDFGs are not always minimal for their equivalent SDFGs [SGB06a]. Throughput-constrained buffer minimization algorithms are also proposed in [CZ12, SOH11, RS14]. However, these techniques leave space for improvement, particularly as graph transformation techniques like retiming and unfolding are not considered.



(a) An example SDFG.



(b) A constrained SDFG obtained by [ZGBS14].



(c) A constrained SDFG obtained by an optimal solution.

Figure 3.3: An example illustrating the limitation of [ZGBS14].

By using retiming and unfolding, the original graph can be transformed to an equivalent graph which has the potential to get better scheduling results. For example, for the SDFG in Figure 3.1, the minimal buffer usage for rate-optimal scheduling is 10. However, after retiming the 4 initial tokens on channel e_1 to 1 token on channel e_2 and 2 tokens on channel e_3, and then unfolding the graph using an unfolding factor of 2, the minimal buffer usage for rate-optimal scheduling is decreased to 8. Although retiming and unfolding are both useful techniques, extra steps have to be taken to get the retiming operations and unfolding factors. As shown in [ZGBS12] and [ZGBS14], self-timed execution (STE) based scheduling can perform retiming and unfolding automatically during its state exploration process. STE is widely used to solve SDFG scheduling problems [ZGBS14, SGB06a, GGS$^+$06, ZGBS12, GBS05, SBGC07].

Solutions for buffer usage minimization for rate-optimal schedules are given in [ZGBS14, SGB06a]. Memory-constrained STE was firstly adopted in [SGB06a] to explore trade-offs in buffer usage and throughput constraints for SDFGs. However, initial tokens are viewed as real data and affect the lower bound of the buffer usage, and therefore the minimal buffer usage is usually not possible to obtain. In addition, the work in [SGB06a] is not specific for buffer size minimization for rate-optimal scheduling. Although the minimal buffer usage for rate-optimal schedules is included in the output of the algorithm, the algorithm has to spend a large amount of time to find the buffer sizes for non-maximal throughput schedules. Thus, the speed of this algorithm is comparatively slow. The work in [ZGBS14] develops a heuristic to solve the problem, where the effect of the initial tokens is eliminated. This heuristic tries to get the minimum buffer size of channels one by one, which may lead to a sub-optimal solution for the total buffer size of all channels. As mentioned in [ZGBS14], their heuristic cannot guarantee the achievement of the minimal buffer sizes.

An example is shown in Figure 3.3, where Figure 3.3a is the original SDFG. The critical cycle of the SDFG is $A \rightarrow D \rightarrow C \rightarrow A$ with an IP of $5/2 = 2.5$. Here, we focus on channels e_1 and e_2 since the heuristic in [ZGBS14] can get minimum buffer size on other channels. As discussed in Section 2.6.1, the buffer size $b(e)$ on a channel without initial tokens can be modelled as a back-pressure edge e' with $it(e') = b(e)$. If the heuristic in [ZGBS14] gets the minimum buffer size for e_1 before e_2, then for rate-optimal scheduling, the minimum buffer size of e_1 is 1. As shown in Figure 3.3b, an edge e'_1 is added to the original SDFG to model the constraint of the buffer size on e_1. Based on the constraint of the buffer size on e_1, when the heuristic in [ZGBS14] tries to minimize the buffer size on e_2, a back-pressure edge e'_2 with 4 initial tokens

has to be added to the original SDFG. This is because the edge e'_2 introduces a new cycle $A \to D \to C \to B \to A$ into the original SDFG. If the number of initial tokens on edge e'_2 is 2, the IP of the new cycle will become $6/2 = 3$, which is larger than the IP of the original SDFG. To get rate-optimal schedules, the IP of the new cycle has to be less than the IP of the original SDFG. Thus, the number of initial tokens on edge e'_2 should be at least 4. This means the minimum buffer size on channel e_2 should be at least 4 for rate-optimal scheduling. On the contrast, for an optimal solution as shown in Figure 3.3c, the channel e_2 can have the minimum buffer size 2 while the buffer size on channel e_1 is increased to 2. The total buffer size of e_1 and e_2 is 4 for the optimal solution, which is 1 less than the total buffer size obtained by [ZGBS14]. In fact, the optimal solution can be obtained by [ZGBS14] by minimizing the buffer size for e_2 before e_1. This example shows how the order of the channels for the buffer size minimization effects the obtained buffer size of the heuristic in [ZGBS14].

3.4 Algorithms

By using an effective retiming method, a state-of-art algorithm proposed in [ZGBS14] is the only work which can eliminate the extra buffer usage introduced by the initial tokens. This algorithm is denoted by ZHU and used as a comparison algorithm in this chapter. BC-STE is jointly used with a binary search process to get the minimum buffer size on channels in sequence. The results of ZHU are affected by the higher bounds of the buffer size and the sequences of the channels during the binary search. As a result, only sub-optimal results can be obtained by the algorithm in many cases. To improve this, we propose an exact searching algorithm and a heuristic, which outperforms ZHU in terms of buffer use.

At first, the proposed exact algorithm narrows down the range of the searching. Then, a solution space searching algorithm is applied to find a buffer distribution d_{min} with the minimal distribution size. The storage dependencies proposed in [SGB06a] are used to accelerate the searching process. Based on the exact algorithm, a heuristic method is also developed by using an approach proposed in [SGB06a] to reduce its execution time. The effect of the initial tokens for buffer usage is eliminated by proposing an extended BC-STE method, which is described next.

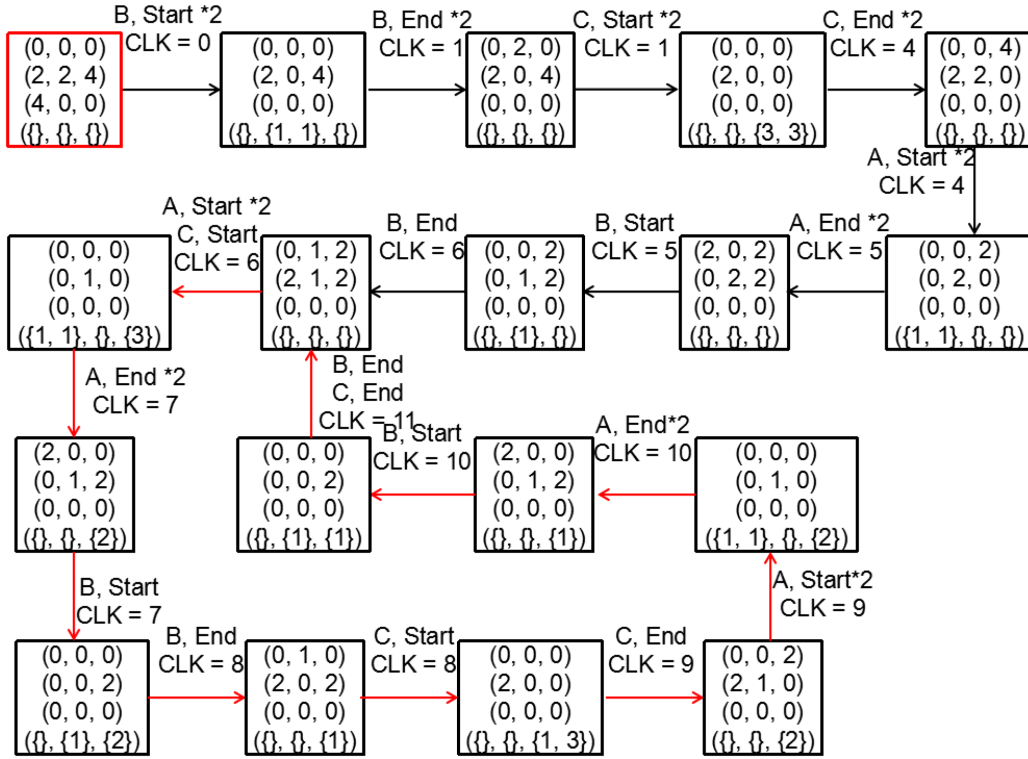


Figure 3.4: The eBC-STE of the example SDFG in Figure 3.1.

3.4.1 Extended Buffer-Constrained Self-Timed Execution

Based on BC-STE, an extended BC-STE (eBC-STE) approach is developed to get the *IP* of an SDFG under the constraint of a buffer distribution without being affected by the initial tokens on the SDFG. Except for the three vectors used in STE, the states of the eBC-STE have an additional vector $\boldsymbol{IT}$ to represent the number of the initial tokens on channels. In this way, the number of initial tokens does not have to be the initial value of $\boldsymbol{T}$ as mentioned in Definition 14, and therefore, does not affect the lower bound of the buffer requirements on channels. Then, the initial value of $\boldsymbol{T}$ is set to a zero vector, and the initial value of an element $\boldsymbol{IT}(e)$ is set to be $it(e)$.

During the eBC-STE process, initial tokens are consumed first. Tokens in $\boldsymbol{T}$ cannot be consumed until all initial tokens are consumed. Initial tokens can only be consumed and cannot be produced. By using the eBC-STE, the nature of the initial tokens is reverted, since they are just used to represent the delays between actors in the SDFGs but not realistic data, and therefore should not occupy any memory space. Similar with the STE in [ZGBS12] and [ZGBS14], eBC-STE also performs retiming and unfolding during the execution process. The retiming can be obtained from the transient phase

and the unfolding factor can be obtained from the periodic phase by using Algorithm 2 in [ZGBS12].

Figure 3.4 is the eBC-STE of the SDFG in Figure 3.1. In Figure 3.4, an extra element $\boldsymbol{IT}$ is introduced between the vector $\boldsymbol{S}$ and $\boldsymbol{R}$ in states, which presents the remaining initial tokens in channels. We can view a transient phase in Figure 3.4, followed by a periodic phase. Since the transient phase only needs to be obeyed once while the periodic phase runs iteratively, the time consumed by the transient phase is negligible. Therefore, the *IP* of the periodic phase is also 5 time units, which is the same as the *IP* of the BC-STE in Figure 3.2. Since $\boldsymbol{IT}$ is not used in Figure 3.2, the number of initial tokens on channel e_1 has to be used as the initial value of the token number on that channel. Therefore to store these tokens, the buffer size on channel e_1 cannot be less than 4. Instead, in Figure 3.4, initial tokens are represented by $\boldsymbol{IT}$; the number of initial tokens no longer affects the buffer size on channels. Therefore, the buffer size on e_1 can be reduced from 4 in Figure 3.2 to 2 in Figure 3.4, while the *IP* of the execution remains the same.

3.4.2 Obtaining the Initial Point

The proposed exact algorithm is a solution-space searching algorithm for buffer distributions. The *initial point* of the searching should be set first. Since the searching is limited to rate-optimal schedules, a good initial point should avoid searching on undesirable solutions, and should be much closer to buffer distributions that enable the existence of rate-optimal schedules rather than the zero point in [SGB06a] or $\max(p,q)$ in [ZGBS14].

Algorithm 4 is used to get the initial point of the searching. In line 1, we get the *LB* of the SDFG to check if a schedule is rate-optimal. If the *IP* of a schedule is equal to *LB*, then it is rate-optimal. The variables d_{LB} and d_{UB} are the lower bound and the upper bound of the buffer distributions on channels. The value of d_{UB} is obtained from the buffer use of the periodic phase of the STE obtained by Algorithm 3. The variable d_{LB} can be obtained according to Section 5 of [ZGBS14]. The variable *BC* is a temporary vector. The variable d_{st} is the initial point to be solved. A property for d_{st} is that each element in d_{st} is the exact lower bound of the buffer size on that channel for rate-optimal schedules. This property can be proved as follows.

Theorem 6. The maximum throughput of an SDFG with a certain buffer distribution does not decrease because of the increase of the buffer size on one or more channels.

Algorithm 4 Obtaining the Initial Point

Input:
An SDFG $G = (V, E)$

Output:
The initial point of the searching d_{st}

Iteration:

1: Get the LB and d_{UB} from Algorithm 3, get d_{LB}
2: **for all** $e \in E$ **do**
3: Perform a binary search over $[d_{LB}(e), d_{UB}(e)]$; assuming x is the value considered, let vector BC be defined as $BC(e) = x$ and $BC(e') = \infty$ if $e' \neq e$; use eBC-STE to get IP and check whether BC is feasible for a rate-optimal schedule and let $d_{st}(e) = BC(e)$ if so.
4: **end for**
5: **return** d_{st}

Proof. For an SDFG with a buffer distribution $[d_0, d_1, ..., d_n]$, if there exists a schedule S that makes the maximum throughput $\mathcal{T}_{max}$, increasing the buffer size of one or more channels does not affect the constraints of the schedule. Then, the schedule S is still applicable to the SDFG, and therefore the maximum throughput of the SDFG is at least to be $\mathcal{T}_{max}$. □

Then, the property for d_{st} can be derived by the following analysis. Assume there exists a storage distribution d_{lst} with a lower buffer requirement on channel e than d_{st} which can still have a rate-optimal schedule. If we increase the buffer size on other channels except for e to ∞, the obtained storage distribution should also have at least one rate-optimal schedule according to Theorem 6. Then the distribution d_{lst} must be reached by the binary search in Algorithm 4. This is a contradiction, hence the assumption is false and it is not possible to have a buffer size on that channel smaller than d_{st}.

3.4.3 Exact Algorithm for Buffer Size Minimization

The exact searching algorithm is shown in Algorithm 5. The initial point of the searching is obtained by Algorithm 4. We use the buffer distribution d_{TUB} obtained by ZHU as a tighter upper bound. This tighter upper bound and a better initial point aim to reduce the search range of the proposed algorithm. Since the algorithms used to obtain the upper bound and the initial point are relatively fast, this search range reducing process is quite time efficient. Its effectiveness is illustrated in Section 3.5.2.

Algorithm 5 Obtaining Minimal Storage Distribution For Rate-Optimal Schedules

---

Input:

An SDFG $G = (V, E)$, iteration bound LB, initial point d_{st} obtained by Algorithm 4, and the storage distribution of the upper bound d_{TUB}

Output:

The minimal storage distribution d_{min}, a schedule of the SDFG under the buffer constraint of d_{min}

Iteration:

1: **if** $|d_{st}| \equiv |d_{TUB}|$ **then**
2:    **return** $d_{min} = d_{TUB}$
3: **end if**
4: Insert initial point d_{st} to a distribution set $dSet$
5: **while** $dSet$ not empty **do**
6:    Set the first distribution in $dSet$ to be the current distribution d
7:    If $|d|$ is greater than or equal to $|d_{TUB}|$, break out of the while-loop
8:    Use eBC-STE to compute current IP and dependency graph $\triangle$ of the graph under current storage distribution constraint
9:    **if** $IP > LB$ **then**
10:       Let SD be the set of storage dependencies in $\triangle$
11:       **for all** channel $e \in SD$ **do**
12:          $d' = d$
13:          $d'(e) = d'(e) + step(e)$
14:          insert d' to $dSet$ in ascending order of distribution size
15:       **end for**
16:       Delete d from $dSet$
17:    **else**
18:       **return** $d_{min} = d$
19:    **end if**
20: **end while**
21: **return** $d_{min} = d_{TUB}$

Procedure 6 Modifications for the Heuristic

```
1: for all channel e ∈ SD do
2:    d′ = d
3:    if the time limit is exceeded then
4:       QF = ⌈(|d_TUB| − |d_st|) ∗ 0.1 ∗ ef⌉
5:       ef++
6:    end if
7:    d′(e) = d′(e) + step(e) ∗ QF
8:    insert d′ to dSet in ascending order of distribution size
9: end for
```

The storage dependency graph introduced in [SGB06a] is used to accelerate the search. Only the channels included in the dependency graph need to be increased within the for-loop, or we have to traverse all channels in the SDFG. The effectiveness of this technique is illustrated in [SGB06a]. The while-loop in Algorithm 5 is used to do the search and only stops when the upper bound $|d_{TUB}|$ is reached or a minimal buffer distribution is found. If $|d_{TUB}|$ is reached, the algorithm returns d_{TUB} as the minimal buffer distribution, since all distributions which have distribution sizes less than d_{TUB} cannot produce a rate-optimal schedule.

3.4.4 A Heuristic for Buffer Size Minimization

If the execution time of the scheduling algorithm is limited and the exact algorithm cannot finish within some time constraints, the heuristic technique in [SGB06a] can be used. According to [SGB06a], the execution time can be exponentially decreased by enlarging the $step(e)$. Therefore, a quantization factor QF can be used to multiply the $step(e)$ if a time limit is exceeded for an SDFG. Then, a heuristic algorithm can be obtained by simply replacing lines 11 to 15 in Algorithm 5 by Procedure 6. The initial values for QF and ef are both set to 1. Large values of QF can significantly reduce the execution time since they enlarge the steps of the search and ignore some solutions between two steps. However, if QF is too large, which means too many solutions are ignored, then the quality of the results will be decreased. Line 4 of Procedure 6 proposes an approach to compute large values of QF which is used in this thesis. Finding a more efficient approach to get QF is an interesting problem for further research.

Table 3.1: The graph generation parameters.

	actor (nr)	degree (avg/var/min/max)	rate (avg/var/min/max/rVS)	initialTokens (prop)	execTime (avg/var/min/max)
Benchmark15	15	3/1/1/5	3/1/1/5/100	0	50/250/10/100
Benchmark30	30	3/1/1/5	12/144/1/24/400	0	50/250/10/100
Benchmark60	60	3/1/1/5	48/2,304/1/96/800	0	50/250/10/100

3.5 Evaluation

3.5.1 Preliminaries

In this evaluation, ZHU is used as the comparison algorithm. The algorithm in [SGB06a] is not used since it is too time-consuming and generally performs worse than ZHU according to the experimental results given in [ZGBS14]. The proposed algorithms and ZHU are implemented within SDF3 [SGB06b] and tested on a virtual machine with the Ubuntu system on a PC running Windows 7 with an Intel i7-2670QM CPU and 8GB RAM. Four cores and 2.6GB RAM are allocated to the virtual machine. A 30-minute time limit for all algorithms is set, since the same time limit is also applied in the work we compare against [ZGBS14].

Using eight SDFGs *bipatite*, *buffercycle*, *example*, *fig8*, *h263decoder*, *modem*, *samplerate* and *satellite* from the "Explore throughput/storage-space trade-offs" benchmark of [SGB06b] and the SDFGs of an H.263 encoder, an H.263 decoder, a granule-level MP3 decoder, a block-level MP3 decoder and an MP3 playback application also from [SGB06b] (Download/Examples section), we compared the proposed exact algorithm and the proposed heuristic with ZHU. The exact algorithm cannot finish within the 30-minute time limit for *fig8* and the block-level MP3 decoder. For all other SDFGs the proposed exact algorithm produces equal results with ZHU in buffer size. The same is also true for the proposed heuristic, which, however, results in a buffer size of 1 less than ZHU for *fig8* (53 versus 54) and 10 less for the block-level MP3 decoder (1,858 versus 1,868). To compare and analyze the proposed algorithms in more detail, we used three groups of randomly-generated SDFGs using SDF3. The three groups are denoted by Benchmark15, Benchmark30 and Benchmark60. Each group contains 1,000 SDFGs with 15, 30, or 60 actors, respectively. The graph generation parameters used are shown in Table 3.1, where *nr* decides the number of actors, *degree* decides the number of output channels of actors, *rate* decides the token producing and consuming rate of actors, *initialTokens* decides the possibility that initial tokens are added to channels, *execTime* decides the execution time of actors, and

Table 3.2: Result improvement and runtime increase for randomly generated SDFGs.

	equal cases	runtime increase (AVG/MED)	improved cases	runtime increase (AVG/MED)	improvement (AVG/MAX/MIN)	completion rate
Benchmark15	83.80%	15.40/1.80	11.80%	182.39/1.94	4.84%/51.26%0.04%	95.60%
Benchmark30	72.60%	6.54/1.93	16.00%	37.99/2.00	5.83%/56.50%/0.05‰	88.60%
Benchmark60	66.50%	2.55/1.92	18.90%	7.39/2.00	5.58%/54.28%0.01‰	85.40%

rVS (under degree) is the abbreviation of repetitionVectorSum which decides the sum of the minimum repetition number of all actors. The detailed explanation of the parameters is given in the website of SDF3 [SGB06b].

3.5.2 Results

The Exact Algorithm

The result improvement and runtime increase of the proposed exact algorithm compared with ZHU is shown in Table 3.2. In the table, the column *equal cases* shows the percentage of cases for which the two algorithms produce the same results, the column *improved cases* shows the percentage of cases for which the proposed algorithm produces better results than ZHU, the columns *runtime* next to *equal cases* and *improved cases* give the average/median multiple of the runtime increase of the proposed algorithm compared with ZHU. The column *improvement* shows the average/maximum/minimum percentages of reduction for the buffer requirements produced by the proposed algorithm for the improved cases. The column *completion rate* shows the percentage of the cases that can finish within the time limit. According to the table, most cases can finish within the time limit. For Benchmark15, 95.60% of the cases have finished within the time limit, in which the proposed algorithm achieves better results for 11.80% of the cases and for other cases gets the same buffer requirements compared with ZHU. On average, the runtime of the proposed algorithm increases 15.40 times for the equal cases and 182.39 times for the improved cases compared with ZHU. However, the median values of the two categories of the cases are both close to 2, which means the runtime increase for most cases is not that high.

According to the results for Benchmark30 and Benchmark60, it appears that the completion rate decreases with the size of the graphs, while the percentage of improved cases increases. This may be because the solution space gets more complicated as the scale of the graphs increases. Therefore, the performance of heuristic algorithms like ZHU is degraded, while the performance of the proposed exact algorithm is not affected

Table 3.3: Result improvement and runtime increase for the proposed heuristic algorithm.

	equal cases	runtime increase (AVG/MED)	improved cases	runtime increase (AVG/MED)	improvement (AVG/MAX/MIN)	completion rate
$QF = 1$						
Benchmark15	83.80%	15.40/1.80	11.80%	182.39/1.94	4.84%/51.26%0.04%	95.60%
Benchmark30	72.60%	6.54/1.93	16.00%	37.99/2.00	5.83%/56.50%/0.05‰	88.60%
Benchmark60	66.50%	2.55/1.92	18.90%	7.39/2.00	5.58%/54.28%0.01‰	85.40%
$QF = \lceil(\lvert d_{TUB}\rvert - \lvert d_{st}\rvert) \times 0.1\rceil$						
Benchmark15	1.60%	249.31/59.27	2.60%	113.28/7.81	10.44%/24.92%0.55%	4.20%
Benchmark30	4.00%	35.01/11.00	7.10%	16.86/3.93	10.33%/49.86%/0.04‰	11.10%
Benchmark60	3.70%	15.04/4.43	9.10%	6.82/2.71	9.96%/54.07%0.02‰	12.80%
$QF = \lceil(\lvert d_{TUB}\rvert - \lvert d_{st}\rvert) \times 0.2\rceil$						
Benchmark15	0.20%	33.71/33.71	0	—	—	0.20%
Benchmark30	0.30%	7.47/6.55	0	—	—	0.30%
Benchmark60	1.80%	5.31/4.53	0	—	—	1.80%
TOTAL						
Benchmark15	85.60%	19.81/1.81	14.40%	169.92/2.05	5.85%/51.26%0.04%	100.00%
Benchmark30	76.90%	8.02/1.94	23.10%	31.49/2.23	7.21%/56.50%/0.04‰	100.00%
Benchmark60	72.00%	3.26/1.94	28.00%	7.21/2.04	7.00%/54.28%0.01‰	100.00%

by the size of the solution space, even though its runtime becomes longer. According to Table 3.2, the runtime increase is still limited for most cases in Benchmark30 and Benchmark60.

The Heuristic Algorithm

To balance execution time with the quality of scheduling, a heuristic algorithm was proposed in Section 3.4.4. The heuristic keeps increasing the value of QF when the time limit is exceeded during the execution of the heuristic. This makes it possible to find solutions for all the cases in the experiment when QF has increased to the value $\lceil(\lvert d_{TUB}\rvert - \lvert d_{st}\rvert) \times 0.2\rceil$. Compared with ZHU, the result improvement and runtime increase of the heuristic are shown in Table 3.3. There are four parts in Table 3.3. The first three parts give the outcome when QF equals to 1, $\lceil(\lvert d_{TUB}\rvert - \lvert d_{st}\rvert) * 0.1\rceil$ and $\lceil(\lvert d_{TUB}\rvert - \lvert d_{st}\rvert) * 0.2\rceil$, respectively. The last part, *TOTAL*, combines the data of the above parts and gives the total results of the heuristic. The values of *equal cases*, *improved cases* and *completion rate* in the second and third parts of Table 3.3 are much lower than the corresponding values in the first part of Table 3.3, as most cases can finish when the quantization factor is 1 and are not counted in parts 2 and 3 in Table 3.3.

We define cases that cannot finish by the proposed exact algorithm within the time limit as *overtime cases*. These cases are solved by the heuristic when QF has increased

Table 3.4: Search Range Reduction.

	Benchmark15	Benchmark30	Benchmark60
Average Reduction	97.02%	95.52%	94.24%
Equal Rate	69.20%	53.30%	37.50%

to $\lceil(|d_{TUB}| - |d_{st}|) * 0.1\rceil$ or $\lceil(|d_{TUB}| - |d_{st}|) * 0.2\rceil$. From part 2 and 3 of Table 3.3, we notice that more than half of the overtime cases can be improved by the proposed heuristic compared with ZHU, and the average improvement of the buffer requirements is higher than the data in Table 3.2. This means most overtime cases are cases which have the potential to be optimized, and therefore they demand more runtime to do the search in their solution spaces.

The Effectiveness of the Search Range Reducing Process

This chapter has proposed Algorithm 4 as a better initial point for search in Algorithm 5 and has used the approach of ZHU as a tighter upper bound of the search, so that the solution space (search range) is reduced. In this experiment, the effectiveness of this method is evaluated. As shown in Table 3.4, the *reduction* of the solution space is calculated by the formula

$$\frac{|d_{UB}| - |d_{TUB}| + |d_{st}| - |d_{LB}|}{|d_{UB}| - |d_{LB}|}.$$

We can see that the solution space is greatly reduced by the proposed search range reducing process.

The *equal rate* is the proportion of the cases which satisfy the condition $|d_{TUB}| = |d_{st}|$. This means that the condition in line 1 of Algorithm 5 is met and the algorithm terminates. Then, the runtime of the exact algorithm only contains the time consumed by Algorithm 4 and ZHU which is used to get the d_{TUB}. For Benchmark15, the equal rate is close to 70% but this value decreases as the size of the graphs increases. This suggests that the search range reducing process works better on small scale graphs, and the effectiveness of this process decreases as the scale of the graphs increases.

3.6 Summary

This chapter has presented an exact algorithm and a heuristic to solve the buffer minimization problem for rate-optimal schedules of SDFGs. By traversing the solution

space, the proposed exact algorithm can produce minimal buffer requirements. Efficient accelerating methods are introduced to decrease the runtime of the traversing. Compared with a state-of-the-art algorithm in [ZGBS14], the proposed algorithm achieves the same or less buffer use for a small set of realistic applications and a larger set of randomly generated SDFGs. A detailed experimental analysis using randomly generated SDFGs demonstrated the improvements achieved by the proposed exact algorithm and heuristic. The buffer size minimization approaches proposed in this chapter is used as **Step 1** of the proposed scheduling framework in Section 1.5.

Chapter 4

Communication-Aware Scheduling for SDFGs

4.1 Overview

Compile-time scheduling strategies are widely used for the estimation of the performance of an application or the generation of deterministic schedules for an application [ABEDK16]. A periodic schedule is necessary for the iterative execution of an SDFG. However, there are two major challenges when scheduling SDFGs. Firstly, the data producing and consuming rates of actors in an SDFG can be different, which means the execution rates of actors in the SDFG can vary. For example, in Figure 4.1, actors A, B and C have to execute 1, 2 and 2 times within one iteration of the SDFG, respectively. Secondly, cycles may exist in an SDFG, as it is the case in Figure 4.1. To deal with these two challenges, a *transformation based* scheduling (or blocked scheduling [LM87b]) methodology can be used. This methodology converts an SDFG to a Directed Acyclic Graph (DAG), and then applies DAG scheduling algorithms [CJSZ08] to deal with the converted DAG. The obtained schedule for the converted DAG is used as a periodic schedule for the original SDFG. However, a graph

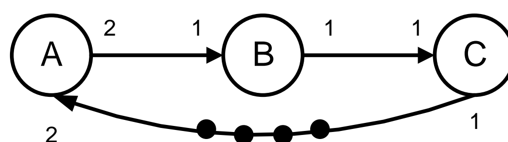


Figure 4.1: An example SDFG.

transformation process (i.e., SDFG to DAG) is required for this kind of scheduling methodology, a process that can be a major part of the runtime of the scheduling. In addition, the size of the obtained DAG can increase dramatically compared with the original SDFG, making the solving time of the scheduling problem even longer. Another problem is that the converted DAG represents only one iteration of the original SDFG. This means that the potential of utilizing parallelism across different iterations of an SDFG is not exploited.

To avoid the drawbacks of the transformation based method, some work (e.g., [ZGBS16, SGB06a, ZGBS12, ZGBS14, BLMB13, LGE12, BMKdD12]) tries to handle SDFGs directly without any conversion. A simple way to do this is to model the problem of scheduling an SDFG as a problem that can be solved with existing solvers. For example, the scheduling problem can be modelled as a linear programming (LP), an integer linear programming (ILP), a constrained programming (CP) or a model-checking problem; representative work that follows each of these approaches to solve the problem can be found in [BMKdD12], [LGE12], [BLMB13] or [MG13], respectively. Although the LP problem can be solved in polynomial time, the work in [BMKdD12] does not consider as a constraint the number of processing cores. The approaches based on ILP, CP and model-checking can model an NP-complete problem. However, the runtime of these approaches increases exponentially with the scale of the problem, which limits their applicability when large SDFGs have to be scheduled.

On the other hand, Self-Timed Execution (STE) stemming from [GGS+06] appears to be both efficient and effective in solving SDFG scheduling problems as shown in a number of papers [ZGBS16, SGB06a, ZGBS12, ZGBS14]. When STE is used as a scheduling method, it simulates the execution of an SDFG by using a state transformation based expression. The *state* of a system is a series of variables which represent its execution state. Changes during the execution of a system are represented by appropriate state transformation. The transforming states are recorded until a periodic execution pattern is found, which leads to a periodic schedule. A static schedule can be obtained by using the starting time of actors within the periodic execution pattern. The non-periodic execution phase before the periodic pattern is called a *transient phase*. The transient phase of STE actually forms a retiming process[ZGBS12]. A *retiming* technique [LS91] enables the execution of some actors before the beginning of the periodic execution phase, something that redistributes the initial tokens in the original SDFG. The periodic pattern may contain multiple iterations of an SDFG, a feature that unfolds the original SDFG.

The retiming and unfolding features embedded in STE are always helpful in improving the performance of STE scheduling algorithms [ZGBS12, ZGBS14, ZGBS16]. In [GGS+06], STE shows the ability to analyze the Maximum Cycle Mean (MCM) of an SDFG efficiently; this determines the maximum throughput of the graph. STE based scheduling is used in [SGB06a] to explore the trade-offs in buffer requirements and throughput for SDFGs. For SDFG scheduling, STE is adopted in [ZGBS14, ZGBS16, SGB06a] to decrease the buffer use of the scheduling results and in [ZGBS12, ZGBS16] to improve the processor utility of the scheduling results. However, in all this work communication between cores is not taken into account.

To the best of the author's knowledge, STE has not been used in prior work to deal with the problem of scheduling SDFGs on multicore systems *with communication delays between the cores*. As shown in [WLW+10], effective retiming can improve the throughput of an SDFG by overlapping the inter-actor communication time with the computing time of actors of the application. Considering that the transient phase in STE can lead to effective retiming of the original SDFG [ZGBS14, ZGBS12, ZGBS16], we argue that STE is potentially suitable for scheduling problems considering communication delays as it allows us to overlap computation with communication. Thus, in this chapter, we use STE based algorithms to propose communication-aware scheduling solutions for SDFGs. The throughput of a system is expected to be enhanced by the retiming in the STE process while no graph transformation process is required.

To provide a solution to the SDFG communication-aware scheduling problem, two questions need to be investigated further:
Q1: Does STE still work well when inter-core communication time has been taken into consideration? Compared with a multicore system that does not consider communication time between cores, an extra set of variables is required to describe the communication states of a multicore system with communication delays. An extra set of variables will imply an increase in the number of states; then, the periodic execution pattern becomes harder to find. Furthermore, for STE, the execution rule for actors is to execute an actor as-soon-as-possible (ASAP). This rule works well when communication delays are ignored. However, no previous work shows or proves that STE scheduling can still find easily periodic schedules when communication delays are taken into consideration. Regardless of whether the rule of executing an actor ASAP can work, this raises a second interesting question.
Q2: Can execution rules other than the ASAP rule be applied to the STE process and give better results? The scheduling process of STE is similar to list scheduling

for DAGs. For list scheduling of DAGs, executing an actor ASAP is called Earliest Ready Task (ERT) scheduling [LHCA88]. Besides the ASAP rule used in ERT, the execution rules used in some of the most successful list scheduling algorithms like Dynamic Level Scheduling (DLS) [SL93] or Heterogeneous Earliest Finish Time (HEFT) [THW02] may be potentially better alternatives of the ASAP rule for STE and may deserve further investigation.

In this chapter, both questions **Q1** and **Q2** are answered. The contributions of this chapter are:

- A generic communication-aware STE (CA-STE) scheduling approach;
- A total of four execution rules (including ASAP) are applied to the generic CA-STE approach to propose four CA-STE scheduling algorithms for homogeneous multicore systems;
- The proposed algorithms for homogeneous systems are further extended to support heterogeneous systems;
- A thorough experimental investigation assesses the proposed algorithms providing answers to **Q1** and **Q2** and showing that the proposed CA-STE scheduling algorithms generally outperform two representative list scheduling algorithms to schedule DAGs.

In the remainder of the chapter, the basic concepts and a problem description are given in Section 4.2. Then, a motivational example to illustrate the advantages of an STE based approach over the transformation based approach is given. A generic CA-STE approach leading to the proposed four algorithms for homogeneous systems and two algorithms for heterogeneous systems is described in Section 4.4, followed by an experimental evaluation in Section 4.5. Finally, Section 4.6 concludes the chapter.

4.2 Background

The SDFG related notations used in this chapter are listed in the first section of Table 4.1. All notations are self-explanatory or explained in Chapter 2. The input SDFGs for the proposed scheduling algorithms are assumed to be sample-rate-consistent and deadlock-free.

Table 4.1: Basic notations.

	SDFG Related
α	An actor in set V
$IN(\alpha)$	The set of input channels of the actor α
$OUT(\alpha)$	The set of output channels of the actor α
$ET(\alpha)$	The execution time of the actor α
$\Upsilon(\alpha)$	The minimum repetition number of the actor α
e	A channel in set E
$\alpha_{src}(e)$	The source actor of the channel e
$\alpha_{dst}(e)$	The destination actor of the channel e
$\boldsymbol{IT}(e)$	The number of initial tokens on the channel e
$TS(e)$	The data size of one unit token on the channel e
$p(e)$	The token producing rate of the source actor of the channel e
$q(e)$	The token consuming rate of the destination actor of the channel e
N	The number of actors in an SDFG
L	The number of channels in an SDFG
SR	The sum of the minimum repetition number of all actors
	Hardware Related
c	A processing core in C
$\boldsymbol{B}(e)$	The FIFO buffer on channel e
b	A token block in the FIFO buffer $\boldsymbol{B}(e)$, $b = (num, et, cid)$
$b(num)$	The number of tokens in the token block b
$b(et)$	The existing time of the token block b
$b(cid)$	The processing core which has produced the token block b
M	The number of processing cores
BW	The bandwidth of the bus
	Objective Related
$RT(G)$	A retiming scheme of an SDFG G
$S(G)$	A periodic schedule of an SDFG G
$IP(S)$	The period of a periodic schedule $S(G)$
$J(S)$	The unfolding factor of a schedule $S(G)$
$TH(S)$	The throughput of a schedule $S(G)$

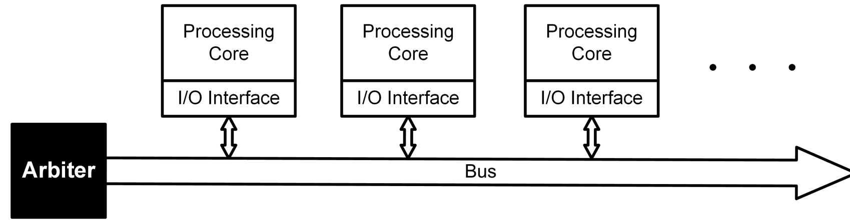


Figure 4.2: The bus-based multicore architecture.

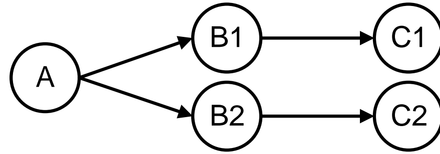
4.2.1 Hardware Architecture

The architectural paradigm we use in this chapter is shown in Figure 4.2. The processing cores are homogeneous. All the processing cores within a multicore processor are connected to a shared bus. Any core can communicate with all other cores through the bus. The bus can only support the communication of one pair of cores at a time, which means once the bus is occupied by some communication, all other inter-core communications are blocked. Data transfer on the bus is managed by a bus arbiter. We assume that the communication operations from cores do not interrupt the computation on the cores. This enables the overlapping of computation and communication. The communication data is modelled as tokens. The duration of communication is calculated by Equation 4.1.

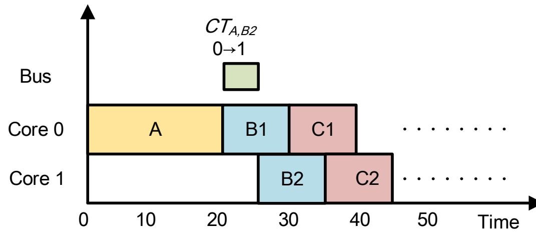
$$T_{com}(e,n_t) = \left\lceil \frac{TS(e) \times n_t}{BW} \right\rceil, \tag{4.1}$$

where $T_{com}(e,n_t)$ is the duration of communication to transfer n_t tokens of size $TS(e)$ between two different cores and BW is the bandwidth of the bus. The duration of communication between two actors on the same core is assumed to be zero.

The processing cores are represented by a vector C. Other hardware related notations are given in the second section of Table 4.1. As the tokens produced by the source actor of a channel may not be consumed immediately, the tokens produced by multiple execution of the source actors may accumulate on the channel. To guarantee the correctness of the execution of an SDFG, the destination actor of the channel has to consume the earliest produced tokens first. Therefore, token transfer on a channel is modelled using a first-in-first-out (FIFO) buffer on the channel. The basic elements on buffers are token blocks. A token block on a channel is produced by one execution of the source actors on the channel.



(a) The converted DAG for the SDFG in Figure 4.1.



(b) An optimal schedule for the converted DAG.

Figure 4.3: The transformation based scheduling.

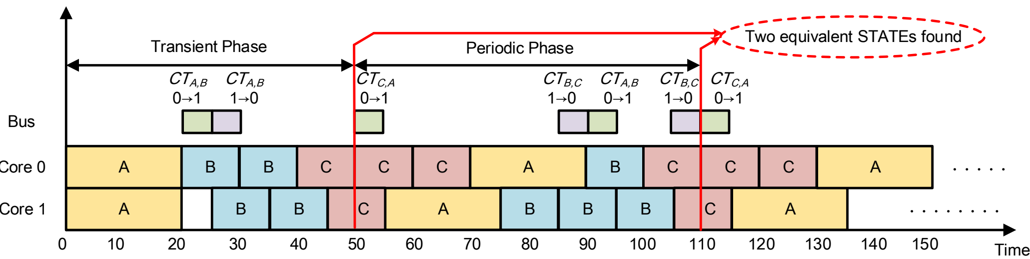


Figure 4.4: The scheduling process of ERAS for the example SDFG in Figure 4.1.

4.2.2 Objective and Criteria

The notations related to the objectives of the scheduling problem are listed in the third section of Table 4.1. The objective of a communication-aware SDFG scheduling algorithm is to obtain a periodic schedule $S(G)$ of an SDFG G. If a retiming process exists, the retiming scheme $RT(G)$ should also be output as part of the result. The criteria to assess the obtained schedules are their throughput, TH, and the runtime of the scheduling algorithms. Note that $S(G)$ obtained by STE may contain the execution of more than one iteration of an SDFG, which is equivalent with unfolding the original SDFG multiple times. The number of iterations in $S(G)$ is called *unfolding factor*, which is denoted by $J(S)$. Although unfolding an SDFG G increases the size of the SDFG by $J(S)$ times, the average period of one iteration can always be decreased. In this chapter, the throughput of a schedule $S(G)$ is calculated by Equation 4.2 as follows:

$$TH(S) = \frac{J(S)}{IP(S)}. \tag{4.2}$$

4.3 Motivational Example

In this section, we use the SDFG in Figure 4.1 to show the advantages of the STE based approach compared to the transformation based approach. The SDFG in Figure 4.1 is assumed to be scheduled onto a two-core system with a bus bandwidth of 1 (token size per time unit). The execution time of each actor A, B and C is 20, 10 and 10 time units, respectively. The token size on all channels is 5.

For the transformation based approach, an SDFG has to be converted to a DAG, and then a scheduling algorithm for DAGs may be used to schedule the converted DAG. The converted DAG for the SDFG in Figure 4.1 is shown in Figure 4.3a, where actors B and C both have two copies $B1$, $B2$ and $C1$, $C2$, respectively. An optimal schedule for the DAG in Figure 3 is shown in Figure 4.3b. The finishing time of this schedule is 45 time units. When this schedule is used for the periodic execution of the original SDFG, the throughput of this schedule is $1/45 \approx 0.022$.

In contrast to the transformation based approach, the STE based approach directly uses the original SDFG to carry out scheduling. A schedule for the SDFG in Figure 4.1 obtained by ERAS, one of the proposed STE based scheduling algorithms in the next section, is shown in Figure 4.4. A transient phase and a periodic phase can be found in this schedule. The transient phase is only performed once at the beginning of the schedule, while the periodic phase is then iteratively used for the following execution of the SDFG in Figure 4.1. The throughput of the schedule is decided by the period of the periodic phase. The period of the periodic phase is 60 time units, and there are two iterations of the SDFG in Figure 4.1 in one periodic phase. Thus, the throughput of the schedule in Figure 4.4 is $2/60 \approx 0.033$, which is 50% higher than the throughput of the optimal transformation based scheduling approach.

The throughput improvement is gained from two techniques: retiming and unfolding. The transient phase of the schedule in Figure 4.4 is a retiming process, which enables some actors to be executed before the periodic phase, turning some intra-iteration data dependencies into inter-iteration data dependencies. The inter-iteration data dependencies do not matter when scheduling one iteration of an SDFG. Furthermore, inter-iteration data communication is much easier to overlap with actor execution in a single iteration as opposed to finding overlaps for intra-iteration

data communication. On the other hand, there are two iterations in the periodic phase of the schedule in Figure 4.4, which unfolds the original SDFG with an unfolding factor of 2. The unfolding technique makes the utilization of parallelism across multiple iterations of an SDFG possible, which is always helpful to improve throughput.

As a result of the existence of the retiming and unfolding process in STE based scheduling, the throughput obtained is expected to be higher than the throughput obtained by transformation based approaches. In addition, STE based approaches do not need to use the SDFG-to-DAG conversion process, which is time-consuming and may enlarge exponentially the size of the original SDFG [SBGC07]. In what follows, we use STE based approaches to propose algorithms for the communication-aware scheduling problem for SDFGs on multicore systems.

4.4 Algorithms

STE based scheduling is a greedy strategy essentially similar to ERT scheduling for DAGs [LHCA88], which always schedules the actor with the earliest starting time in each scheduling step. The difference between ERT scheduling and STE based scheduling is that the latter does not remove an actor from the scheduling list even after all invocations of this actor in one iteration have been scheduled. This allows the overlapping of multiple iterations of an SDFG's execution, something that enables the consideration of retiming and unfolding in STE based scheduling. The STE process stops when a periodic execution pattern is found.

In this section, we first extend the original STE to address the problem of communication-aware scheduling for SDFGs. After that, four scheduling algorithms are proposed based on four actor-to-core allocation rules. Lastly, two of the proposed homogeneous algorithms are extended to support heterogeneous multicore systems.

4.4.1 Communication-Aware Self-Timed Execution

Firstly, we give the description of the generic CA-STE scheduling approach proposed in this chapter. The proposed scheduling algorithms are based on this generic CA-STE scheduling approach. The STE process is a simulation of the execution of an SDFG on a multicore system. The execution process is modelled as a transformation of *STATEs* of the system. Different from previous work [ZGBS16, SGB06a, ZGBS12, ZGBS14], which does not consider communication delays and ignores the interconnection

architecture, for the proposed communication-aware scheduling, the state of communication is a significant part of the STATE of the whole system.

For the bus-based architecture adopted in this chapter, a state is described as a tuple $STATE = (\boldsymbol{B}, \boldsymbol{R}, \boldsymbol{IT}, BUS)$. $\boldsymbol{IT}$ and $\boldsymbol{B}$ are two vectors of channels as shown in Table 4.1. As explained in Section 4.2.1, each element $\boldsymbol{B}(e)$ in $\boldsymbol{B}$ is a FIFO buffer. A token block b in the buffer $\boldsymbol{B}(e)$ is a tuple (num, et, cid), where $b(num)$, $b(et)$ and $b(cid)$ are all explained in Table 4.1. R is a vector of cores in which each element $\boldsymbol{R}(c)$ is a list of tuples (aid, rt), where aid is an actor currently allocated on core c and rt is the remaining time of the execution of the actor aid on core c. The list $\boldsymbol{R}(c)$ is sorted in ascending order of rt. BUS is a list of unoccupied time slots on the shared bus. We use $slot = (st, le)$ to represent one time slot in BUS, where st is the starting time of $slot$ and le is the duration of $slot$. The list BUS is sorted in ascending order of $slot(st)$.

Compared with the state for STE defined in Definition 14, the state for CA-STE is represented by more complex variables. The vector $\boldsymbol{T}$ representing the number of tokens on channels in Definition 14 is replaced by the vector $\boldsymbol{B}$ which gives more detailed information about the location and existing time of token blocks. The vector $\boldsymbol{R}$ representing the remaining execution time of actors in Definition 14 becomes a vector representing the remaining execution time of actors on corresponding cores. The vector $\boldsymbol{IT}$ used in Definition 16 is also adopted in the state for CA-STE. An extra variable BUS is introduced to describe the state of the bus.

We are now in a position to describe the generic CA-STE scheduling algorithm, shown in Algorithm 7. Line 1 of the algorithm simply initializes a STATE list SL and a current STATE $CurState$. One thing to notice is the initial value of BUS is a list with only one $slot$, where $slot(st) = -\infty$ and $slot(le) = +\infty$. The loop body in the algorithm keeps simulating the execution of an SDFG on a given hardware architecture under a certain actor-to-core allocation rule. The current STATE $CurState$ is changed in lines 4-7. The past STATEs of the system are recorded in the list SL in line 3. If the current STATE $CurState$ equals to a STATE in SL, this means the system will keep repeating the STATE sequence from the element equal to $CurState$ to the end of the list SL in the execution that follows. Therefore, this sequence can be used as a periodic schedule, while the sequence before the periodic sequence is the retiming scheme. The equivalent condition of two STATEs is given in Definition 17.

Definition 17. Two STATEs are equivalent if and only if all the elements in the tuples $(\boldsymbol{B}, \boldsymbol{R}, \boldsymbol{IT}, BUS)$ of the two STATEs are equal.

In line 4 of Algorithm 7, *free actors* are the actors which have enough input tokens

Algorithm 7 Generic CA-STE scheduling

Input:

An SDFG $G = (V, E)$, a vector of cores C, and the bandwidth of the bus BW;

Output:

A retiming scheme $RT(G)$ and a periodic schedule $S(G)$;

Iteration:

1: Define SL as an empty list of $STATE$ and $CurState$ as the initial STATE of the system;
2: **while** $CurState \notin SL$ **do**
3: Insert $CurState$ to the end of the list SL;
4: Get all the free actors and insert them into a list FA;
5: Use an actor-to-core allocation rule in Section 4.4.2 to decide the allocation of the actors in FA to the processing cores; use Algorithm 8 to perform the actor-to-core allocation operation;
6: Finish the execution of all end-ready actors;
7: Clock proceeds;
8: **end while**
9: **return** $RT(G)$ as the sequence in SL from the start of SL to the element which equals to $CurState$, and $S(G)$ as the sequence in SL from the element which equals to $CurState$ to the end of SL;

to execute at least once. We give the definition of free actors in Definition 18.

Definition 18. An actor α is a free actor if and only if

$$\forall e \in IN(\alpha), \sum_{b \in \boldsymbol{B}(e) | b(et) \geq 0} b(num) \geq q(e) + \boldsymbol{IT}(e).$$

The actor-to-core allocation rule in line 5 of Algorithm 7 is the key for an STE scheduling algorithm. Different rules are introduced in Section 4.4.2. In the current section, we only introduce the operation of allocating an actor to a core, which is shown in Algorithm 8. The first for-loop in Algorithm 8 (lines 2-16) is used to get the earliest arriving time of all the input tokens for one execution of an actor α on a core c. In the meantime, the input tokens are also consumed from their buffers. Then, in line 18, the earliest available time of core c is compared with the earliest arriving time of the input tokens to get the earliest starting time for the actor. After that, the remaining time of the execution of the actor on c is inserted to the end of the list $\boldsymbol{R}(c)$. In the second for-loop (lines 20-22), the token blocks to be generated are appended at the end of the buffers of the output channels of actor α with a negative existing time, which indicates the remaining time of the token blocks to be generated.

Algorithm 8 Actor-to-core allocation

Input:

The current STATE $CurState = (\boldsymbol{B}, \boldsymbol{R}, \boldsymbol{IT}, BUS)$, an actor α and a core c;

Output:

The earliest ready time t_{er} of α on c;

Iteration:

1: Define t_{er} as the earliest starting time of actor α on core c and initialize t_{er} as 0;
2: **for all** $ch \in IN(\alpha)$ **do**
3: Define $tcnt$ as the number of tokens that has been counted and initialize $tcnt$ as 0;
4: **while** $tcnt < q(e)$ **do**
5: Set first element of $\boldsymbol{B}(e)$ as b, and define the number of tokens to transfer as $tnum$;
6: **if** $\boldsymbol{IT}(e) > 0$ **then**
7: $tcnt \leftarrow \min(q(e), \boldsymbol{IT}(e))$;
8: $\boldsymbol{IT}(e) \leftarrow \boldsymbol{IT}(e) - tcnt$;
9: **end if**
10: $tnum \leftarrow \min(b(num), q(e) - tcnt)$;
11: $b(num) \leftarrow b(num) - tnum$; $tcnt = tcnt + tnum$;
12: If $b(num) = 0$, pop the first element b from $\boldsymbol{B}(e)$;
13: Get the earliest arriving time of the tokens to be transfered as t_t using Algorithm 9;
14: $t_{er} \leftarrow \max(t_{er}, t_t)$;
15: **end while**
16: **end for**
17: Get the longest remaining time in $\boldsymbol{R}(c)$ as rt; if $\boldsymbol{R}(c) = \emptyset$, set $rt = 0$;
18: $t_{er} \leftarrow \max(t_{er}, rt)$;
19: Set $rt' \leftarrow t_{er} + ET(\alpha)$ and insert (α, rt') to the end of the list $\boldsymbol{R}(c)$, where rt' is the remaining execution time of α on core c;
20: **for all** $ch \in OUT(\alpha)$ **do**
21: Insert $b = (num, et, cid)$ to the end of the list $\boldsymbol{B}(e)$, where $b(num) = p(e)$, $b(et) = -rt'$, and $b(cid) = c$;
22: **end for**
23: **return** t_{er};

Algorithm 9 Get the earliest arriving time

Input:

The token block of the tokens transferred from b; the number of the tokens to be transferred *tnum*; and the core to transfer to c;

Output:

Earliest arriving time, t_t;

Iteration:

1: **if** $b(cid) = c$ **then**
2: $t_t \leftarrow -b(et)$;
3: **else**
4: Get communication duration $cd \leftarrow \lceil tnum * TS(e)/BW \rceil$;
5: **for all** $slot \in BUS$ **do**
6: Set $sed \leftarrow slot(st) + slot(le)$ and $ted \leftarrow -b(et) + cd$;
7: **if** $(slot(st) \leq -b(et)$ **and** $sed \geq ted)$ **or** $(slot(st) > -b(et)$ **and** $slot(le) \geq cd)$ **then**
8: $t_t \leftarrow \max(slot(st), -b(et) + cd)$;
9: Remove the time period in *slot* occupied by the communication; if the communication occupied a middle period in *slot*, then *slot* should be split into two separate slots in *BUS*;
10: **break**;
11: **end if**
12: **end for**
13: **end if**
14: **return** t_t

In line 13 of Algorithm 8, the earliest arriving time is hardware-dependent. For the bus-based interconnection, the earliest arriving time can be obtained by Algorithm 9. As indicated by lines 1 and 2 of Algorithm 9, if the destination core c is the same as the core where the token block b exists, the communication duration is 0, and then the earliest arriving time t_t is $-b(et)$, where $-b(et)$ is the earliest possible starting time of the communication. Otherwise, the communication duration is obtained by Equation 4.1 assigning a value to cd in line 4. The for-loop from line 5 to line 12 in Algorithm 9 is used to find a suitable time slot on the bus for the communication. The communication period must be contained in the slot. Then, the earliest arriving time is $t_t = \max(slot(st), -b(et)) + cd$.

In line 6 of Algorithm 7, all *end-ready* actors are removed from R of $CurState$. The definition of end-ready actors is given in Definition 19.

Definition 19. For a tuple $(\alpha, rt) \in \boldsymbol{R}(c)$, if $rt = 0$, the actor α is end-ready on core c.

The last step in the while-loop in Algorithm 7 is clock increase ('clock proceeds'). The clock step is the minimum time step that can change the $STATE$ of the system. For the actors on cores, the minimum clock step is

$$Step_\alpha = \min_{\forall r \in R | r \neq \emptyset} \big(r_0(rt)\big),$$

where r_0 is the first element in the list r. While for tokens on channels, the minimum step is

$$Step_{ch} = \min_{\forall e \in E | \boldsymbol{B}(e) \neq \emptyset} \left(\min_{\forall b \in \boldsymbol{B}(e) | b(et) < 0} \big(-b(et) \big) \right).$$

Then, the minimum clock step of the whole system is obtained as $clk = \min(Step_\alpha, Step_{ch})$. All the time-related variables in $CurState$ change as the clock proceeds, including

- $\forall c \in C | \forall r \in \boldsymbol{R}(c)$, $r(rt) = r(rt) - clk$;
- $\forall e \in E | \forall b \in \boldsymbol{B}(e)$, $b(et) = b(et) + clk$;
- $\forall slot \in BUS$, $slot(st) = slot(st) - clk$.

For the bus-based communication, another important operation in the clock increase step is to remove unnecessary time slots in BUS. Let $b(et)_{max}$ be the longest existing time of all token blocks on buffers of the channels. Then, time slots earlier than $b(et)_{max}$ will never be used during the subsequent execution process and therefore can be removed. The removing operation is $\forall slot \in BUS$,

- if $slot(st) + slot(le) \leq -b(et)_{max}$, remove $slot$ from BUS;
- else if $slot(st) < -b(et)_{max}$, set $slot(le) = slot(le) - b(et)_{max} - slot(st)$ and $slot(st) = -b(et)_{max}$, in this order.

If the condition $CurState \notin SL$ of the while-loop in Algorithm 7 is not met, then a periodic $STATE$ list is found. As shown in line 9 of Algorithm 7, a periodic schedule $S(G)$ and a retiming scheme $RT(G)$ are obtained at last.

4.4.2 Scheduling Algorithms

Based on the generic CA-STE scheduling algorithm shown in Algorithm 7, we construct four scheduling algorithms by using four actor-to-core allocation rules in line 5 of Algorithm 7. The four scheduling algorithms are: **earliest ready actor scheduling**, **earliest finishing actor scheduling**, **multiple allocation earliest ready actor scheduling** and **multiple allocation earliest finishing actor scheduling**.

Earliest ready actor scheduling (ERAS) uses the ASAP rule of the ERT algorithm for DAG scheduling [LHCA88]. The earliest ready time of an actor α on a core c can be obtained by Algorithm 8 (when Algorithm 8 is only used to get the earliest ready time for an actor, the current STATE $CurState = (\boldsymbol{B}, \boldsymbol{R}, \boldsymbol{IT}, BUS)$ should not be varied in the algorithm). Communication delays have been taken into consideration to get the earliest ready time of actors. However, only unoccupied cores can be used during the actor allocation process. We refer to this kind of allocation strategy as *multiple allocation disabled scheduling* (MADS). Here, we define a core c as unoccupied if $\boldsymbol{R}(c) = \emptyset$, otherwise it is occupied.

The allocation process of ERAS is given as follows.

- Step 1: Select a pair of a free actor α and an unoccupied core c with the earliest ready time among all possible combinations of free actors and unoccupied cores;
- Step 2: Allocate α on c, remove α from FA and label c as an occupied core;
- Step 3: Repeat the former two steps until no free actor or unoccupied core exists.

As an example, the SDFG in Figure 4.1 is used to illustrate the scheduling process of ERAS in Figure 4.4. In Figure 4.4, $CT_{A,B}$ indicates the communication time between actor A and B on the bus. The $0 \rightarrow 1$ under $CT_{A,B}$ means that the communication is from core 0 to core 1. The SDFG executes according to the execution rule of ERAS as shown in Figure 4.4. A periodic phase is obtained after a transient phase. The transient

phase forms the retiming process of the SDFG; and the periodic phase can then be used as a periodic schedule.

Earliest finishing actor scheduling (EFAS) adopts a rule similar to ERAS except that instead of the earliest ready time, the earliest finishing time of actors on cores is used as the scheduling priority. The earliest finishing time t_{ef} of an actor α on core c can be obtained by

$$t_{ef} = t_{er} + ET(\alpha). \tag{4.3}$$

By replacing the earliest ready time with the earliest finishing time in Step 1 of the allocation process of ERAS, the allocation process of EFAS is obtained. EFAS also belongs to the MADS family.

For DAGs, a scheduling algorithm called DLS, proposed in [SL93], enables the multiple allocation of actors on a core. This kind of allocation strategy is referred to as *multiple allocation enabled scheduling* (MAES) in this chapter. MAES allows the free actors to be allocated on all cores in C including the occupied cores (by allocating them after the execution of existing actors on the core). As shown in [SL93], this rule can improve the performance of list-based scheduling for DAGs as the communication time is better handled compared with the MADS rule. Based on this fact, we can expect that applying MAES rules in STE based approaches for SDFGs can also result in better performance than the traditional MADS rule ASAP. In this chapter, two MAES algorithms for SDFGs are proposed: **multiple allocation earliest ready actor scheduling** (MERAS) and **multiple allocation earliest finishing actor scheduling** (MEFAS). MERAS and MEFAS are the MAES versions of ERAS and EFAS, respectively. The allocation process of MERAS/MEFAS is given below.

- Step 1: Select a pair of a free actor α and a core c with the earliest ready time (for MERAS) or earliest finishing time (for MEFAS) among all possible combinations of free actors and all cores;
- Step 2: Allocate α on c, remove α from FA and label c as an occupied core;
- Step 3: Repeat the former two steps until no free actor exists.

The time complexity of the STE scheduling algorithms is dependent on the SDFG. As shown in Algorithm 7, the end situation of the while-loop is that a $STATE$ in list SL is found to be equal with $CurState$. As mentioned in [GGS+06], two equivalent $STATE$s are guaranteed to be found since the number of $STATE$s for a system is finite. Fortunately, in practice, for STE scheduling without consideration of communication

delays, the runtime is always much faster than the worst-case of traversing all possible $STATE$s as shown in [GGS+06]. When communication time is taken into consideration, the number of STATEs of a system increases by some orders of magnitude, although the number of STATEs is still finite. If the runtime of STE scheduling becomes longer than existing methods by orders of magnitude as the number of STATEs explodes, the answer to **Q1** is 'no' and STE-based scheduling methods are not suitable for communication-aware scheduling of SDFGs. In Section 4.5, we use detailed experimental evaluation to answer this question.

4.4.3 Extension for Heterogeneous Systems

We further extend the CA-STE based scheduling methodology to systems with heterogeneous processing cores. In a heterogeneous system, the execution time of an actor on different types of cores can be different. We use $ET_c(\alpha)$ to represent the execution time of actor α on core c. For a highly heterogeneous system where some cores are designed as accelerators for specific functions, an actor may not be able to execute on a certain core type. In this case, $ET_c(\alpha)$ is set to $+\infty$.

Different from a system with homogeneous cores where the execution time of an actor on all the cores is the same, for a system with heterogeneous cores, the execution time of actors is decided by the cores they are mapped on. Therefore, only using the earliest ready time of actors to decide the actor-to-core allocation may lead to a significantly long execution time of an SDFG. For example, if an actor α has the smallest ready time on a core c, but the execution time of the actor on that core is $ET_c(\alpha) = +\infty$, then according to the rule of ERAS, the actor α should be mapped on c, which makes the execution time of the system to be $+\infty$. Therefore, only actor-to-core allocation rules using finishing time of actors as a criterion can be used for systems with heterogeneous cores, which means the rule of executing an actor ASAP is not applicable for these systems. On the other hand, among the proposed algorithms in this chapter, two algorithms for homogeneous systems EFAS and MEFAS can be extended to algorithms for heterogeneous systems.

Based on EFAS and MEFAS, by using Equation 4.4 to calculate the finishing time of an actor α on a core c instead of using Equation 4.3, we can get two heterogeneous scheduling algorithms HEFAS and HMEFAS, respectively.

$$t_{ef} = t_{er} + ET_c(\alpha) \tag{4.4}$$

Compared with homogeneous multicore systems, the heterogeneity of cores further increases the possible number of STATEs during the STE process of an SDFG. Therefore, in theory, a periodic execution pattern is harder to be found for the STE on heterogeneous multicore systems. In Section 4.5, we also use experimental results to answer **Q1** and **Q2** for the two STE based heterogeneous algorithms.

4.5 Evaluation

4.5.1 Settings for Homogeneous Algorithms

In this section, we evaluate the performance and runtime of the four proposed STE scheduling algorithms by comparing them with two representative communication-aware DAG scheduling algorithms DLS and EFT [CJSZ08]. EFT is a homogeneous version of the HEFT algorithm proposed in [THW02]. The runtime of all algorithms is obtained on a workstation with an Intel Xeon E5-2667V3 processor and 32 GB RAM. All the algorithms in the experiment are implemented and evaluated within SDF3 [SGB06b]. A time limit of 10 minutes is applied to all the algorithms in the experiment to stop the execution of algorithms that run for too long. In practice, this applies only to DLS and EFT, which have difficulty to build a schedule in some cases that will be explored next. In most other cases, the runtime of each algorithm is less than 1 second.

All SDFGs in the experimental evaluation are assumed to run on a bus-based homogeneous multicore processor. The processor runs at 500MHz and the bandwidth of the bus in the processor is 8GB/s. The parameters of the processor are similar to the multicore DSP processor TMS320C6472 made by TI.

The same data set used in [GTA06] is used in the experimental evaluation, which consists of 12 realistic applications. The 12 realistic applications are all obtained from the folder streamit-src-2.1/apps/benchmarks/asplos06 of Streamit 2.1.1 (Older release) [Str18]. They are denoted by BeamFormer, BitonicSort, ChannelVocoder, DCT, DES, FFT, FilterBank, FMRadio, MPEG2Decoder, Serpent, TDE and Vocoder. For each one of the 12 SDFGs, the number of actors, N, and the sum of the minimum repetition number of actors, SR, are shown in the Table 4.2.

The function *transformSDFtoHSDF(SDFgraph* G)* in SDF3 [SGB06b] is used to do the SDFG-to-HSDFG conversion for EFT and DLS. The runtime of this conversion for the 12 applications is given in the last column of Table 4.2. The 12 realistic applications are written in StreamIt language, where a filter can be viewed as an actor. Thus, an

Table 4.2: The number of actors N, the sum of the minimum repetition number of actors SR and the runtime of the SDFG-to-HSDFG conversion for each of the 12 realistic SDFGs used in the experiments

Application	N	SR	Runtime of SDFG-to-HSDFG Conversion (ms)
BeamFormer	56	106	5.61
BitonicSort	40	64	2.32
ChannelVocoder	55	1,835	378.39
DCT	8	1,058	110.43
DES	53	991	216.77
FFT	17	1,406	837.73
FilterBank	85	512	34.72
FMRadio	43	47	0.83
MPEGdecoder	23	2,177	1,081.73
Serpent	120	6,355	6,071.52
TDE	29	5,027	46,569.01
Vocoder	114	404	25.06

application can be extracted as an SDFG through the functions provided in StreamIt, including the estimation for the execution time of actors in clock cycles and for the token size on channels in bytes. The push and pop rates of filters are used as the token producing and consuming rates of actors, respectively. Besides push and pop, there is another operation called peek in StreamIt language which is used to read the tokens on buffers without popping them out from buffers. For a filter, if its peek rate pk is higher than its pop rate q, one invocation of the filter actually requires pk tokens as its input. On the other hand, the peek rate cannot be used as the token consuming rate of an actor, or the behaviour and the sample-rate-consistency of an SDFG would be changed. Therefore, to correctly represent the amount of the data required by one invocation of an actor while preserving the correctness of the application, instead of changing the token consuming rates, the token size on channels is modified as $TS(e) = TS'(e)pk(\alpha_{dst})/q(\alpha_{dst})$, where $TS'(e)$ is the original token size on a channel e. StreamIt language allows the existence of internal variables in an actor, which means one invocation of an actor may depend on its former invocations. An actor with internal variables is referred to as a *stateful* actor in [GTA06]. For a stateful actor, one self-edge with one initial token is added to the actor. The token producing and consuming rates of the self-edge are both 1; and the token size of the self-edge is the total size of the internal variables in bytes.

If an SDFG is not a strongly connected graph, which means some actors in the graph

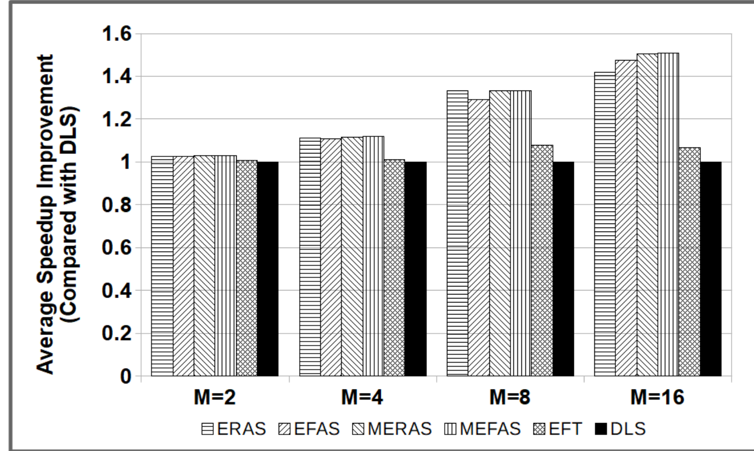


Figure 4.5: The average speedup improvement of all algorithms compared with DLS using 2, 4, 8 and 16 cores for the 8 realistic applications which can be completed by all homogeneous algorithms.

only have input channels or output channels, the graph should be converted to a strongly connected graph before the STE scheduling process, as mentioned in [ZGBS14].

4.5.2 Results for Homogeneous Algorithms

Two criteria are used for the evaluation of the algorithms in the experiment. The first criterion is throughput speedup of an algorithm: this is the throughput obtained using multiple cores divided by the throughput of a single core baseline. The second criterion is the runtime of an algorithm. The throughput of an SDFG $G = (V, E)$ on a single core is calculated by

$$TH_{single}(G) = \frac{1}{\sum_{\alpha \in V} ET(\alpha) \times \Upsilon(\alpha)}.$$

Then, if the throughput obtained by a schedule S for an SDFG G using M cores is $TH_M(S)$, the speedup is

$$\frac{TH_M(S)}{TH_{single}(G)}.$$

For the runtime, to smooth out minor deviations between runs, we run each algorithm ten times for each application and take the average runtime of these executions.

The speedup and runtime obtained by all the algorithms on 2, 4, 8, and 16 cores for the 12 realistic SDFGs are given in Table 4.3. Since EFT and DLS cannot finish within the 10-minute time limit for DES, MPEGdecoder, Serpent and TDE, the speedup and runtime for these cases are labeled as "—" in Table 4.3. The runtime of an SDFG-to-HSDFG conversion is also included in the runtime of EFT and DLS.

Table 4.3: Speedup and runtime for all homogeneous algorithms and applications using 2, 4, 8 and 16 homogeneous cores.

Application	#Core	Speedup						Runtime (ms)					
		ERAS	EFAS	MERAS	MEFAS	EFT	DLS	ERAS	EFAS	MERAS	MEFAS	EFT	DLS
BeamFormer	2	2.00	2.00	2.00	2.00	2.00	2.00	292.24	211.56	231.00	296.48	29.16	31.09
	4	3.90	3.92	4.00	4.00	4.00	4.00	180.36	454.89	551.16	280.70	26.87	32.83
	8	7.74	7.47	7.99	7.99	7.52	6.90	719.83	800.56	1,764.02	3,110.96	30.72	35.06
	16	15.79	16.00	15.79	15.78	14.74	10.43	1,645.03	228.57	19,651.63	204,190.20	30.26	44.19
BitonicSort	2	2.00	2.00	2.00	2.00	1.74	1.75	22.97	29.18	45.94	76.97	34.62	35.70
	4	3.99	3.93	3.85	3.93	2.30	2.29	172.89	105.52	50.65	193.16	33.33	38.15
	8	6.44	6.43	6.27	6.26	2.30	2.29	81.17	91.87	52.87	36.67	35.53	33.44
	16	6.65	6.65	6.88	6.88	2.30	2.29	42.76	42.80	30.23	30.34	35.14	37.47
ChannelVocoder	2	2.00	2.00	2.00	2.00	2.00	2.00	553.05	1,886.87	940.83	947.78	593.54	8,488.73
	4	4.00	3.99	4.00	4.00	4.00	4.00	3,027.56	2,481.72	909.18	895.99	607.70	17,356.45
	8	7.97	7.97	7.98	7.98	7.98	7.98	2,691.03	3,157.68	1,246.14	1,223.88	708.88	26,609.37
	16	15.87	15.87	15.87	15.87	15.87	15.87	894.65	2,520.65	1,376.70	1,371.91	805.68	47,650.74
DCT	2	2.00	2.00	2.00	2.00	2.00	2.00	52.39	48.60	50.25	50.03	1,467.67	5,172.41
	4	3.99	3.99	3.99	3.99	3.99	3.99	40.03	34.13	44.36	44.63	1,355.44	5,143.43
	8	7.96	7.96	7.96	7.96	7.96	7.96	49.79	40.62	61.69	62.38	1,499.87	5,362.73
	16	15.81	15.81	15.81	15.81	15.81	15.81	49.74	40.48	71.33	71.72	1,496.86	6,400.86
DES	2	1.91	1.91	1.91	1.91	—	—	206.89	199.30	164.32	159.71	—	—
	4	3.48	3.49	3.49	3.50	—	—	146.26	153.83	135.01	141.87	—	—
	8	5.91	5.93	5.92	5.95	—	—	168.66	169.47	156.19	165.47	—	—
	16	9.02	9.07	9.09	9.14	—	—	182.16	183.92	181.87	189.89	—	—
FFT	2	1.95	1.95	1.99	1.99	1.99	1.99	77.15	76.21	33.00	33.34	22,383.59	29,154.48
	4	3.72	3.72	3.91	3.91	3.91	3.91	68.05	68.83	36.24	36.55	20,476.58	34,850.04
	8	6.77	6.77	7.41	7.41	7.41	7.41	69.08	69.60	48.56	48.98	22,301.08	44,025.81
	16	11.56	11.56	13.02	13.02	13.02	13.02	72.02	72.10	70.25	71.01	21,658.22	60,794.70
FilterBank	2	2.00	2.00	2.00	2.00	2.00	2.00	199.45	195.48	162.23	159.71	60.14	175.96
	4	4.00	4.00	4.00	4.00	4.00	4.00	152.64	122.24	151.48	155.69	52.76	201.84
	8	8.00	7.99	8.00	8.00	8.00	8.00	152.14	110.96	186.09	189.39	56.75	237.87
	16	15.96	15.01	15.97	15.97	15.97	15.97	160.22	167.91	214.41	212.27	60.99	533.76
FMRadio	2	2.00	2.00	2.00	2.00	1.89	1.83	358.53	25.46	79.44	89.85	1.69	1.83
	4	3.88	3.88	3.88	3.88	3.29	3.15	333.33	359.05	36.26	54.20	1.56	2.13
	8	7.58	6.39	7.28	7.31	5.99	4.13	93.16	31.20	41.18	41.93	1.81	2.41
	16	12.32	15.15	15.24	15.24	5.99	5.98	65.16	371.17	289.74	307.85	1.83	3.70
MPEGdecoder	2	1.99	2.00	2.00	2.00	—	—	617.85	586.14	477.66	886.00	—	—
	4	3.97	3.96	3.99	3.99	—	—	1,387.33	951.75	1,218.69	1,238.51	—	—
	8	7.78	7.90	7.92	7.92	—	—	2,000.78	1,190.88	2,858.84	2,879.68	—	—
	16	14.76	14.95	15.00	15.00	—	—	1,556.25	1,205.73	1,888.42	1,995.08	—	—
Serpent	2	1.99	1.98	1.98	1.98	—	—	17,136.38	23,718.12	14,005.92	12,616.95	—	—
	4	3.92	3.91	3.92	3.92	—	—	20,211.87	30,763.51	25,600.82	23,369.26	—	—
	8	7.66	7.62	7.66	7.66	—	—	21,309.05	29,217.34	33,801.41	27,977.04	—	—
	16	14.60	14.46	14.57	14.57	—	—	22,512.69	24,160.63	38,437.14	39,119.44	—	—
TDE	2	1.99	1.99	2.00	2.00	—	—	4,957.70	5,661.49	4,729.95	4,727.60	—	—
	4	3.96	3.96	3.98	3.98	—	—	8,248.62	10,055.82	12,900.11	13,256.48	—	—
	8	7.81	7.86	7.91	7.91	—	—	12,675.48	13,542.11	29,923.58	25,234.07	—	—
	16	15.26	15.31	15.50	15.50	—	—	16,782.01	20,681.46	60,632.15	55,403.49	—	—
Vocoder	2	1.92	1.91	1.94	1.94	2.00	1.93	810.81	771.26	688.11	668.47	1,509.20	1,798.89
	4	3.74	3.71	3.78	3.77	3.98	3.78	871.77	852.19	901.88	879.81	1,603.30	1,690.37
	8	6.73	6.72	6.77	6.81	7.39	6.80	792.92	770.28	865.82	901.17	1,700.78	1,742.36
	16	11.20	11.25	11.22	11.24	12.74	11.31	814.71	806.78	979.31	1,008.02	1,786.61	1,794.70

As shown in Table 4.3, the proposed STE based algorithms get equal or better speedup than the transformation based algorithms EFT and DLS in most cases. When the number of cores is small, such as 2 or 4, all the algorithms used in the experiment can get near-optimal speedup in most cases and the advantage of the proposed algorithms is not obvious. When the number of cores is large, such as 8 or 16, the speedup obtained by the proposed algorithms appears to become much higher than the speedup obtained by EFT and DLS (cf. BitonicSort, FMRadio, and to some extent BeamFormer). We also note that, as expected from the discussion in Section 4.4.2, the performance of the proposed MAES algorithms MERAS and MEFAS is generally better than the performance of the proposed MADS algorithms ERAS and EFAS.

For the eight realistic applications which can be completed by all algorithms within the time limit, the average speedup improvement of each algorithm normalized by the speedup of DLS, for each different number of cores, is shown in Figure 4.5. It can be seen that, on average, the proposed algorithms achieve better speedup than DLS and EFT. It appears that the average speedup improvement of the proposed algorithms increases with the number of cores. The two best MAES algorithms, MERAS and MEFAS, achieve on average about 50% higher speedup than DLS when the number of cores is 16.

In terms of the runtime, all the proposed algorithms can finish within the time limit. However, the transformation based algorithms EFT and DLS cannot finish the schedule for four applications within the time limit. In total, as shown in Table 4.3, the runtime of EFT and/or DLS is much higher than the proposed algorithms for eight of the twelve applications, namely, ChannelVocoder, DCT, DES, FFT, MPEGdecoder, Serpent, TDE and Vocoder. As we can see from Table 4.2, the *SR* of ChannelVocoder is 1,835, the *SR* of DCT is 1,058, the *SR* of DES is 991, the *SR* of FFT is 1,406, the *SR* of MPEGdecoder is 2,177, the *SR* of Serpent is 1,835, the *SR* of TDE is 5,027, and the *SR* of Vocoder is 404. Out of these 8 applications, 7 applications have an *SR* value close to or higher than 1,000, a value which is much larger than the values of their number of actors, N. There are also three applications (namely, BeamFormer, BitonicSort and FMRadio) whose runtime is smaller when using EFT and DLS compared to the proposed STE based algorithms. The *SR* of BeamFormer is 106, the *SR* of BitonicSort is 64 and the *SR* of FMRadio is 47, values which are around or less than 100, and are close to the values of N for these applications. This suggests that the runtime of EFT and DLS is significantly affected by the value of *SR* of an SDFG. This is because the SDFG-to-DAG conversion increases the number of actors from N in the original SDFG to *SR* in the converted DAG. Therefore, if the value of *SR* is much higher than N, the runtime of DAG scheduling

algorithms becomes much longer than the runtime of the proposed algorithms which directly use the original SDFGs for the scheduling.

For the four applications having a low EFT or DLS runtime (less than 500 ms), the runtime of the SDFG-to-DAG conversion constitutes a significant part of the overall runtime. For example, when the number of cores is 2, for BeamFormer, FilterBank and FMRadio, the SDFG-to-DAG conversion occupies up to 18%, 66%, and 53% of the total runtime of EFT and DLS, respectively. A special case is BitonicSort, which also has a short runtime, but the SDFG-to-DAG conversion takes less than 10% of the total runtime. This is because the values of N and SR of BitonicSort are close to each other, which means the SDFG-to-DAG conversion for BitonicSort is relatively simple, hence the runtime of the conversion is low. For the remaining eight applications considered in the experiments, the percentage of the runtime consumed by the SDFG-to-DAG conversion is less than 10% of the total runtime of EFT and DLS.

Finally, we note that the runtime of the MAES algorithms is longer than the runtime of the MADS algorithms in most cases when the number of cores is 16. This is because MAES algorithms have to check the earliest ready or finishing time of free actors on all cores while MAES algorithms only need to check the unoccupied cores. A special case is that for BeamFormer, the runtime of MEFAS becomes significantly longer than other cases when 16 cores are used. This implies the complex actor-to-core allocation rules used in MAES algorithms could cause exponential increase in the number of STATEs in the STE process and dramatically prolong the runtime of MAES algorithms for some experiment cases.

4.5.3 Settings and Results for Heterogeneous Algorithms

The 12 realistic applications are also used to test the proposed heterogeneous algorithms for systems using heterogeneous cores. Heterogeneous EFT (HEFT) [THW02] and heterogeneous DLS (HDLS) [SL93] are used as baselines. Algorithms ERAS and MERAS, which only consider the earliest ready time of actors during their actor-to-core allocation processes, are also tested in this experiment section to show the advantages of HEFAS and HMEFAS for heterogeneous systems. The heterogeneous multicore processor used in this section is assumed to have two different types of processing cores T1 and T2, and the number of these two types of cores is the same on all heterogeneous systems in the experiment. Under this assumption, three different situations for the heterogeneity for processing cores in a multicore system are examined in this experiment. For these three situations, the execution time of an actor on T1 cores is set as the execution time

Table 4.4: Speedup and runtime for all heterogeneous algorithms and applications using heterogeneous cores with random speed.

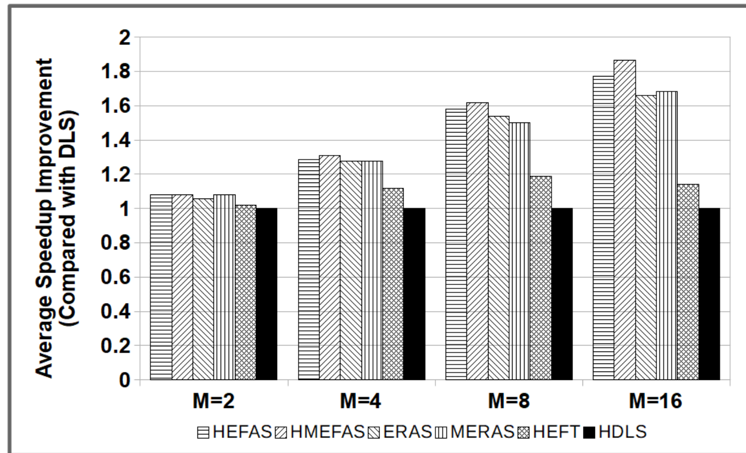
Application	#Core	Speedup						Runtime (ms)					
		HEFAS	HMEFAS	ERAS	MERAS	HEFT	HDLS	HEFAS	HMEFAS	ERAS	MERAS	HEFT	HDLS
BeamFormer	2	2.11	2.15	2.01	2.14	2.05	2.18	367.66	2,018.51	3,710.50	489.84	57.95	65.48
	4	4.07	4.33	3.94	4.03	4.14	4.04	613.79	1,246.12	785.99	895.90	89.52	108.67
	8	7.42	8.80	7.97	7.42	7.21	6.15	411.49	6,834.83	10,212.31	1,442.41	124.25	180.75
	16	15.56	17.48	15.12	15.39	10.67	8.81	747.03	328,031.30	70,228.93	4,112.03	172.59	336.83
BitonicSort	2	2.05	2.09	1.99	2.06	1.71	1.32	137.50	97.90	80.87	75.73	93.61	96.63
	4	3.98	4.16	3.98	3.87	2.33	1.40	551.77	321.53	745.98	390.85	144.34	145.70
	8	6.51	6.31	5.91	5.64	2.47	1.40	117.86	50.46	340.29	186.00	197.77	204.23
	16	7.16	7.14	6.13	5.85	2.47	1.40	95.05	194.46	782.63	52.67	280.28	291.04
ChannelVocoder	2	2.13	2.08	2.06	2.13	2.04	2.19	1,559.74	1,339.92	1,212.85	1,433.01	699.93	12,298.21
	4	4.26	4.15	4.12	4.26	4.07	4.39	2,000.76	1,783.48	2,232.49	1,970.21	837.03	61,858.43
	8	8.47	8.33	8.21	8.46	8.13	8.72	2,495.28	4,152.93	5,830.63	2,588.44	1,026.37	125,716.08
	16	16.76	16.65	16.49	16.98	16.26	17.42	2,588.03	9,767.19	13,598.85	3,268.40	1,344.97	302,440.80
DCT	2	2.01	2.01	2.01	2.00	2.01	2.01	78.30	69.63	63.21	70.87	3,701.75	5,871.92
	4	3.97	3.97	3.97	3.97	3.97	3.97	72.43	71.45	78.99	69.24	5,764.10	12,187.64
	8	7.91	7.92	7.91	7.91	7.92	7.91	86.57	89.51	89.09	63.51	8,151.17	23,220.01
	16	15.49	15.49	15.48	15.49	15.49	15.48	125.29	131.16	165.80	104.20	11,329.98	45,037.93
DES	2	1.93	1.94	1.92	1.92	—	—	199.83	210.84	218.14	199.46	—	—
	4	3.51	3.54	3.40	3.42	—	—	179.25	240.92	288.08	206.76	—	—
	8	6.02	6.08	5.86	5.89	—	—	184.79	266.60	319.28	184.32	—	—
	16	8.96	9.25	8.50	8.44	—	—	224.04	328.52	618.84	271.47	—	—
FFT	2	2.25	2.08	2.08	2.24	2.08	2.19	100.57	50.26	49.33	87.57	62,762.94	66,693.37
	4	4.33	4.07	4.04	4.33	4.07	4.26	84.59	53.46	47.45	81.27	90,557.42	101,814.45
	8	8.15	7.69	7.56	7.99	7.69	7.52	83.30	69.25	60.51	80.22	122,587.65	142,284.08
	16	14.05	13.42	12.95	13.27	13.42	12.59	97.01	115.77	104.10	89.20	176,479.70	236,051.90
FilterBank	2	2.12	2.15	2.13	2.12	2.06	2.18	253.70	205.17	175.26	216.57	58.80	304.78
	4	4.17	4.27	4.26	4.16	4.09	4.43	249.89	217.28	201.11	238.90	68.06	1,025.73
	8	8.26	8.67	8.26	8.24	8.20	8.56	247.58	223.14	290.28	254.86	75.61	2,156.31
	16	15.79	17.57	16.56	15.45	16.28	17.03	251.32	292.38	312.49	302.48	86.74	6,174.72
FMRadio	2	2.19	2.16	2.16	2.19	2.02	1.86	6,366.73	32.63	33.64	6,201.37	1.70	2.87
	4	4.15	4.24	4.21	4.15	3.67	3.02	177.93	590.39	936.15	58.63	1.99	5.73
	8	7.15	7.70	7.61	7.21	5.89	3.98	93.98	491.86	191.52	106.35	2.33	13.37
	16	12.97	15.37	12.84	14.66	5.89	5.60	195.69	6,656.31	1,135.58	4,152.21	2.96	39.81
MPEGdecoder	2	2.06	2.03	2.03	2.05	—	—	633.81	685.52	591.39	676.23	—	—
	4	4.07	4.08	4.07	4.06	—	—	862.30	1,344.42	1,346.34	863.62	—	—
	8	8.08	8.12	8.08	8.06	—	—	981.99	2,744.50	2,592.36	1,159.76	—	—
	16	15.71	15.65	15.28	15.53	—	—	1,184.13	2,942.74	4,974.05	1,739.44	—	—
Serpent	2	2.10	2.04	2.03	2.09	—	—	21,400.55	12,361.04	13,344.95	21,270.28	—	—
	4	4.15	4.03	4.01	4.13	—	—	42,077.26	24,676.19	24,479.81	41,125.55	—	—
	8	8.16	7.89	7.85	8.13	—	—	39,593.34	37,419.19	40,775.98	34,416.51	—	—
	16	15.53	15.14	14.83	15.37	—	—	43,489.95	61,284.24	80,608.03	62,928.23	—	—
TDE	2	2.14	2.06	2.05	2.14	—	—	8,604.65	5,532.26	5,494.64	8,503.81	—	—
	4	4.23	4.12	4.07	4.23	—	—	14,403.96	14,543.83	12,274.35	13,104.25	—	—
	8	8.38	8.24	8.06	8.38	—	—	16,424.22	27,118.77	28,663.97	17,016.80	—	—
	16	16.10	16.37	15.68	15.94	—	—	23,131.16	58,478.55	55,556.35	22,172.91	—	—
Vocoder	2	1.94	2.03	1.97	1.94	2.06	1.99	862.28	874.22	828.84	855.16	3,769.75	3,559.65
	4	3.78	3.81	3.84	3.77	4.08	3.33	897.07	1,084.85	1,045.81	922.25	4,954.14	5,176.28
	8	6.33	6.81	6.24	6.19	7.69	6.46	846.18	1,182.35	1,261.24	907.79	6,440.29	7,102.33
	16	10.47	10.98	10.18	10.61	12.29	10.83	889.31	1,325.78	1,298.47	816.23	9,090.44	11,039.38

Table 4.5: Speedup and runtime for all heterogeneous algorithms and applications using low speed heterogeneous cores.

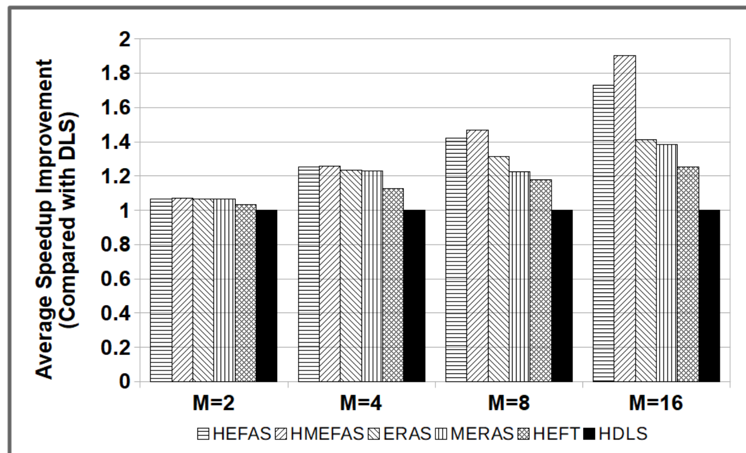
Application	#Core	Speedup						Runtime (ms)					
		HEFAS	HMEFAS	ERAS	MERAS	HEFT	HDLS	HEFAS	HMEFAS	ERAS	MERAS	HEFT	HDLS
BeamFormer	2	2.24	2.25	2.25	2.25	2.23	2.22	374.97	1,678.76	2,404.72	51,769.64	56.03	72.09
	4	4.37	4.50	4.49	4.33	4.28	3.99	492.62	1,516.10	1,588.33	429.24	91.44	97.31
	8	8.61	8.99	8.72	7.47	8.52	8.23	410.65	849.99	1,210.21	378.63	117.39	162.09
	16	15.51	—	14.20	14.70	15.21	11.27	1,569.37	—	12,198.54	280.15	180.29	344.26
BitonicSort	2	2.24	2.24	2.24	2.24	1.84	1.50	121.57	156.67	997.09	149.01	87.70	100.67
	4	4.37	4.22	3.95	4.36	2.61	1.63	315.59	153.43	29.31	1,611.58	148.30	125.47
	8	6.79	7.00	5.42	5.15	3.44	1.71	324.30	111.23	823.04	1,551.06	206.41	188.54
	16	8.71	9.46	5.28	5.34	3.44	1.71	167.11	83.31	514.61	4,802.35	289.13	308.71
ChannelVocoder	2	2.25	2.25	2.25	2.25	2.25	2.25	2,423.03	1,006.94	1,372.48	2,607.62	654.72	8,972.51
	4	4.50	4.50	4.50	4.50	4.50	4.49	3,033.86	1,765.25	1,167.87	2,874.41	850.20	31,824.06
	8	8.97	8.97	8.97	8.97	8.97	8.97	2,808.65	2,583.05	2,242.27	3,575.15	967.07	83,276.20
	16	17.81	17.94	17.93	17.68	17.94	17.94	3,070.96	4,584.09	5,526.63	3,056.81	1,180.90	257,307.30
DCT	2	2.24	2.24	2.24	2.24	2.24	2.24	94.24	71.30	67.21	88.40	3,489.19	4,457.98
	4	4.39	4.42	4.41	4.34	4.42	4.35	90.60	101.57	103.24	110.33	5,889.76	9,725.00
	8	8.74	8.82	8.37	7.87	8.82	8.61	147.18	195.93	179.75	193.87	8,109.62	17,673.57
	16	16.73	16.22	15.61	15.61	16.22	16.22	203.20	425.80	123.56	76.93	12,002.85	34,676.77
DES	2	2.17	2.18	2.10	2.11	—	—	186.84	197.26	193.23	184.65	—	—
	4	4.00	4.06	3.78	3.66	—	—	172.22	221.40	238.36	179.87	—	—
	8	7.07	7.08	6.42	6.40	—	—	206.00	301.95	289.06	186.58	—	—
	16	10.08	11.28	9.42	9.37	—	—	216.51	329.10	361.35	196.66	—	—
FFT	2	2.23	2.23	2.23	2.17	2.23	2.22	78.75	55.05	47.05	79.38	52,402.76	59,310.24
	4	4.30	4.42	4.28	4.00	4.42	4.38	68.00	48.46	54.91	78.33	87,316.06	84,763.85
	8	7.88	8.39	7.64	7.40	8.39	8.13	83.50	83.06	71.19	79.24	108,771.47	122,533.31
	16	14.04	15.44	12.58	11.79	15.44	14.43	88.56	111.19	103.25	75.95	162,637.00	176,719.40
FilterBank	2	2.23	2.25	2.23	2.22	2.25	2.25	219.41	203.78	185.22	200.91	61.68	348.95
	4	4.39	4.48	4.47	4.36	4.48	4.10	205.14	202.74	218.21	208.27	67.62	1,011.63
	8	8.47	8.93	8.75	8.28	8.96	8.03	234.28	235.51	232.70	195.04	80.06	2,268.15
	16	16.63	17.44	17.44	16.65	17.48	16.73	252.66	296.73	268.53	172.42	87.22	5,992.39
FMRadio	2	2.25	2.25	2.25	2.25	2.16	2.14	315.09	56.92	32.98	291.42	1.76	2.42
	4	4.33	4.37	4.40	4.02	3.95	3.76	80.68	248.21	167.67	20.87	1.90	5.40
	8	8.19	8.23	8.44	7.04	6.54	6.04	44.40	108.01	281.14	22.92	2.23	12.28
	16	12.65	16.59	12.36	11.82	8.99	6.16	241.16	315.62	58.20	68.76	2.90	38.23
MPEGdecoder	2	2.24	2.25	2.25	2.23	—	—	617.81	522.85	514.32	608.35	—	—
	4	4.39	4.49	4.50	4.46	—	—	941.75	1,269.04	1,326.33	1,026.60	—	—
	8	8.73	8.89	8.84	8.79	—	—	1,388.17	3,142.60	2,402.57	1,422.31	—	—
	16	17.29	17.29	16.95	16.94	—	—	1,301.70	3,618.02	4,127.20	2,006.22	—	—
Serpent	2	2.24	2.24	2.22	2.22	—	—	22,527.70	12,615.89	11,085.56	20,433.61	—	—
	4	4.41	4.43	4.34	4.35	—	—	33,131.49	22,173.31	24,770.83	39,956.60	—	—
	8	8.57	8.73	8.42	8.27	—	—	40,479.27	35,774.70	40,885.09	34,878.59	—	—
	16	16.53	16.69	15.51	15.28	—	—	23,541.69	41,064.77	60,071.21	39,778.00	—	—
TDE	2	2.25	2.25	2.25	2.24	—	—	4,966.48	4,366.64	4,418.51	4,845.02	—	—
	4	4.47	4.49	4.47	4.45	—	—	9,581.76	12,428.38	10,728.07	8,577.61	—	—
	8	8.84	8.93	8.83	8.75	—	—	14,125.71	27,546.51	27,211.59	12,933.32	—	—
	16	17.45	17.56	17.11	16.95	—	—	19,581.35	54,750.00	47,247.37	16,786.25	—	—
Vocoder	2	2.19	2.21	2.17	2.19	2.22	2.21	781.20	797.58	877.04	868.83	3,824.12	3,429.81
	4	3.92	3.98	4.03	3.90	4.39	3.80	830.69	1,002.42	1,002.60	894.44	4,687.76	4,643.91
	8	6.98	7.43	6.30	6.38	8.21	7.33	843.60	1,057.71	1,033.77	801.07	6,372.73	7,331.86
	16	11.25	11.34	10.53	10.42	13.77	11.70	940.79	1,184.94	1,347.81	788.37	7,725.57	11,861.38

Table 4.6: Speedup and runtime for all heterogeneous algorithms and applications using high speed heterogeneous cores.

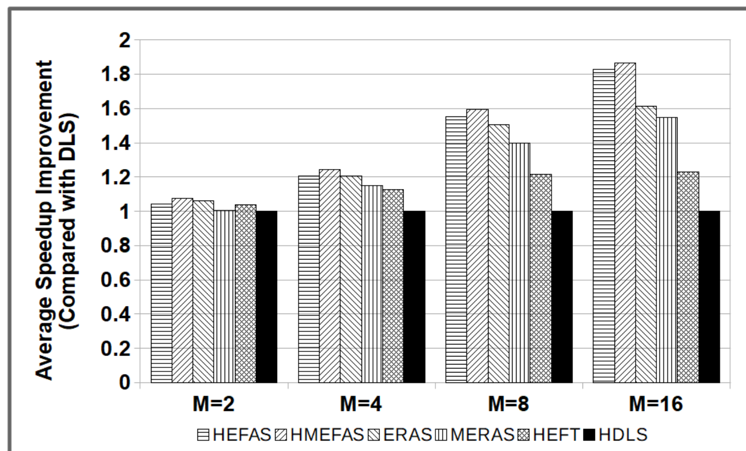
Application	#Core	Speedup						Runtime (ms)					
		HEFAS	HMEFAS	ERAS	MERAS	HEFT	HDLS	HEFAS	HMEFAS	ERAS	MERAS	HEFT	HDLS
BeamFormer	2	2.16	2.18	2.18	2.01	2.17	2.18	279.05	166.94	157.61	116.68	63.89	67.95
	4	4.33	4.35	4.35	3.81	4.26	3.91	328.31	2,023.03	1,307.93	289.62	80.88	87.18
	8	7.94	8.42	8.31	6.68	7.67	5.90	782.45	810.79	2,256.12	177.11	109.74	152.34
	16	15.93	17.23	16.23	14.28	11.49	8.41	1,005.49	6,367.10	23,729.40	930.62	165.83	246.87
BitonicSort	2	1.40	1.52	1.51	1.38	1.46	1.09	31.69	35.61	31.40	23.95	94.49	95.20
	4	2.95	3.05	3.03	2.84	2.36	1.24	53.54	69.22	92.61	32.45	128.38	121.11
	8	5.76	5.68	5.26	5.23	2.72	1.31	660.26	184.99	286.77	135.71	180.81	195.72
	16	7.78	7.69	6.71	6.70	2.72	1.31	62.24	144.12	705.08	396.58	273.70	252.37
ChannelVocoder	2	2.20	2.20	2.20	2.06	2.20	2.20	3,556.46	2,105.95	1,636.46	1,712.00	731.14	20,955.87
	4	4.40	4.40	4.40	4.12	4.40	4.39	3,932.22	2,951.64	1,945.89	1,495.23	771.91	35,160.54
	8	8.46	8.77	8.77	7.97	8.78	8.78	7,281.69	3,237.69	2,664.41	1,984.19	867.97	86,662.69
	16	16.50	17.54	17.54	15.56	17.56	17.56	11,211.15	4,932.17	3,227.51	1,860.71	1,044.06	159,347.10
DCT	2	4.08	4.11	4.10	4.01	4.11	4.00	173.85	76.26	75.61	127.26	3,759.94	8,715.44
	4	8.11	8.19	7.99	7.99	8.19	8.12	203.13	175.66	79.18	113.18	4,983.46	10,925.98
	8	15.80	16.29	15.05	15.05	16.29	16.21	108.02	120.77	95.01	90.84	7,042.14	17,344.71
	16	29.49	32.33	26.98	26.98	32.33	32.04	124.32	170.70	139.46	97.68	10,120.87	29,462.68
DES	2	2.31	2.35	2.26	2.26	—	—	218.22	224.23	190.10	178.41	—	—
	4	4.13	4.33	3.98	3.99	—	—	242.48	290.80	243.50	198.27	—	—
	8	6.90	7.46	6.52	6.50	—	—	224.57	276.05	260.41	177.36	—	—
	16	10.49	11.53	9.66	9.59	—	—	261.50	323.66	472.75	250.07	—	—
FFT	2	4.32	4.84	4.69	4.19	3.88	4.17	101.15	54.38	49.98	86.83	56,545.71	64,230.96
	4	8.39	9.16	8.49	7.91	7.61	8.01	121.27	66.58	56.53	85.08	79,539.73	86,071.88
	8	16.13	16.05	14.20	14.22	14.56	14.02	104.45	75.31	71.02	82.66	111,319.53	122,091.54
	16	28.16	25.95	21.23	22.81	25.49	24.03	108.84	115.19	133.31	71.66	184,436.20	189,749.10
FilterBank	2	1.83	1.86	1.84	1.66	1.87	1.88	318.58	218.67	171.47	199.21	65.87	429.87
	4	3.61	3.77	3.71	3.32	3.74	3.76	439.11	271.81	212.55	224.64	72.64	801.46
	8	6.92	7.54	7.53	6.54	7.47	6.95	410.78	261.66	219.09	215.77	85.11	1,537.81
	16	14.23	15.07	15.04	12.16	15.07	14.28	480.68	298.98	248.52	199.21	97.31	3,947.99
FMRadio	2	2.60	2.57	2.45	2.59	2.59	2.54	52.34	28.11	32.21	36.82	1.82	2.46
	4	4.85	5.16	4.57	4.87	4.37	4.20	129.87	100.94	245.71	348.60	1.86	5.30
	8	9.42	10.20	9.47	7.53	6.52	6.04	44.80	1,176.46	1,571.78	207.34	2.33	11.32
	16	19.78	20.38	12.58	12.24	12.69	11.07	2,419.20	316,153.10	145.90	38.92	2.71	30.57
MPEGdecoder	2	1.88	1.99	1.98	1.83	—	—	1,361.73	1,237.24	1,026.47	324.76	—	—
	4	3.75	3.99	3.90	3.68	—	—	1,829.27	1,444.20	1,197.36	689.00	—	—
	8	7.53	7.95	7.65	7.20	—	—	2,997.63	2,689.68	1,892.91	916.65	—	—
	16	15.52	15.63	14.43	14.20	—	—	3,465.66	3,383.81	2,842.13	640.89	—	—
Serpent	2	1.88	1.98	1.96	1.79	—	—	52,956.31	14,342.19	12,644.14	13,567.94	—	—
	4	3.74	3.95	3.89	3.53	—	—	75,925.72	22,199.66	20,096.83	20,231.07	—	—
	8	7.67	7.84	7.63	6.93	—	—	125,907.10	29,156.86	28,536.86	20,004.58	—	—
	16	14.98	15.48	14.53	13.24	—	—	142,120.94	31,284.39	39,896.33	21,865.83	—	—
TDE	2	4.08	4.89	4.89	3.07	—	—	7,149.21	2,100.99	1,828.69	3,232.53	—	—
	4	8.19	9.73	9.76	6.04	—	—	16,661.78	7,474.90	5,802.92	6,569.57	—	—
	8	15.22	19.19	18.90	10.56	—	—	20,388.67	14,378.83	11,564.02	9,147.95	—	—
	16	30.37	37.11	35.14	18.49	—	—	37,803.95	28,818.11	24,058.08	12,598.63	—	—
Vocoder	2	1.22	1.22	1.22	1.21	1.22	1.22	953.89	928.51	951.09	887.16	3,318.95	3,176.91
	4	2.31	2.34	2.31	2.29	2.43	2.33	1,030.48	1,068.92	884.69	789.60	4,409.31	4,740.65
	8	4.39	4.55	4.38	4.35	4.85	4.22	1,022.88	1,141.87	933.62	830.32	5,858.21	6,009.59
	16	7.92	8.07	7.44	7.42	8.92	7.97	1,040.44	1,181.94	1,014.30	694.01	8,212.06	10,515.45



(a) Using random heterogeneous cores.



(b) Using low speed heterogeneous cores.



(c) Using high speed heterogeneous cores.

Figure 4.6: The average speedup improvement of all algorithms compared with DLS using 2, 4, 8 and 16 cores for the 8 realistic applications which can be completed by all heterogeneous algorithms.

used in Section 4.5.1. All actors are able to be executed on T1 cores. The execution time of actors on T2 cores is given below.

- Random heterogeneous cores (**Situation 1**) – The execution time of an actor on T2 cores is obtained by multiplying a factor by the execution time of the actor on T1 cores. The factor is randomly generated from 0.5 to 1.5 following a uniform distribution. All actors can be executed on T2 cores.
- Low speed heterogeneous cores (**Situation 2**) – The execution time of an actor on T2 cores is set to be double the execution time of the actor on T1 cores. All actors can be executed on T2 cores.
- High speed heterogeneous cores (**Situation 3**) – The execution time of an actor on T2 cores is set to be one tenth of the execution time of the actor on T1 cores. Only actors with even ID numbers can be executed on T2 cores.

In Situation 1, by using random heterogeneous cores, we simulate systems consisting of two types of processing cores which have similar speed but perform differently in processing different actors. Then, in Situation 2, low speed heterogeneous cores are used to model systems which consist of high performance cores and low speed energy-efficient cores. Lastly, in Situation 3, high speed heterogeneous cores are used to simulate systems having dedicated accelerating cores for some actors. We still use speedup and runtime as the criteria for heterogeneous algorithms. For Situation 1 and Situation 2, the throughput of a single core $TH_{single}(G)$ is calculated by

$$TH_{single}(G) = \frac{1}{\sum_{\alpha \in V}(ET_{T1}(\alpha) + ET_{T2}(\alpha))/2 \times \Upsilon(\alpha)}.$$

In Situation 3, since T2 cores cannot process all the actors in an SDFG, $TH_{single}(G)$ is obtained by using the execution time of actors on $T1$ cores only as following.

$$TH_{single}(G) = \frac{1}{\sum_{\alpha \in V} ET_{T1}(\alpha) \times \Upsilon(\alpha)}.$$

Similar to the illustration for the experimental results for homogeneous algorithms, for the Situation 1, 2 and 3, the speedup and runtime of the heterogeneous algorithms for the 12 realistic applications are shown in Tables 4.4, 4.5, and 4.6, respectively; and the average speedup improvements of heterogeneous algorithms for the three Situations are shown in Figure 4.6, where the average speedup improvement of each algorithm is normalized by the speedup of HDLS.

As shown in Tables 4.4, 4.5, and 4.6, for all the three Situations, the proposed heterogeneous algorithms HEFAS and HMEFAS achieve significant improvements for BeamFormer, BitonicSort and FMRadio in terms of speedup compared with HEFT and HDLS. For ChannelVocoder, DCT, FFT, FilterBank and Vocoder, the speedup obtained by HEFAS and HMEFAS is always similar to HEFT and HDLS. The scheduling for DES, MPEGdecoder, Serpent and TDE cannot be finished by HEFT and HDLS within the time limit for all cases in the experiment. For the two algorithms ERAS and MERAS which only consider the earliest ready time of actors during actor-to-core allocation, the speedup obtained by these two algorithms is, in most cases, worse than HEFAS and HMEFAS which use the earliest finishing time of actors to decide actor-to-core allocations, and even worse than HEFT or HDLS for some cases, such as FFT using 16 cores as shown in Tables 4.5 and 4.6 and DCT using 16 cores as shown in Table 4.6. As shown in Figure 4.6, for Situation 1, 2, and 3, HMEFAS achieves the highest speedup on average among the six algorithms used in the experiment. When 16 cores are used, the speedup obtained by HMEFAS is on average 86%, 90% and 86% higher than HDLS in Situation 1, 2 and 3, respectively. The average speedup of ERAS and MERAS becomes much worse than HEFAS and HMEFAS when the number of cores is as large as 8 or 16.

In terms of runtime, the proposed heterogeneous scheduling algorithms HEFAS and HMEFAS have less runtime than HEFT and/or HDLS for 10 of the 12 applications except for BeamFormer and FMRadio in all three Situations. The exponential increase of the runtime of STE based scheduling algorithms can be observed to be more frequent on heterogeneous systems than homogeneous systems, something that can be observed with the runtime of HMEFAS for BeamFormer or FMRadio using 16 cores in Tables 4.4, 4.5 and 4.6. Especially in Table 4.5, HMEFAS cannot finish within the time limit for scheduling BeamFormer using 16 cores. Compared with homogeneous algorithms in Table 4.3, the runtime of scheduling algorithms for heterogeneous systems is in general higher for all the three Situations as shown in Tables 4.4, 4.5 and 4.6. These observations suggest that for STE based scheduling algorithms, the runtime is longer for heterogeneous systems than for homogeneous systems since there are more potential STATEs in the STE process for heterogeneous systems than for homogeneous systems. Nevertheless, the runtime of STE based scheduling algorithms for heterogeneous systems is still much less than the runtime of HEFT and HDLS in most cases of the experiment.

4.6 Summary

In this chapter, a CA-STE approach is proposed to schedule SDFGs on a multicore system with communication delays. Four different actor-to-core allocation rules are used to form four scheduling algorithms for homogeneous systems, where two of the four algorithms are extended to support heterogeneous systems. Through the experimental evaluation, we can answer the two questions raised in the introduction. With respect to **Q1**, STE based scheduling algorithms are still applicable to communication-aware scheduling problems for SDFGs as, in almost all cases (except one case), STE scheduling algorithms can finish within the time limit in the experiment. For **Q2**, two MAES scheduling rules have been successfully embedded to the generic STE scheduling approach and were tested with higher performance than the traditional MADS rule ASAP. On average, the proposed STE scheduling algorithms always outperform DLS and EFT (or their heterogeneous versions HDLS and HEFT on heterogeneous systems) in the experiment in terms of throughput. The runtime of the proposed algorithms is much lower than DLS and EFT (or HDLS and HEFT on heterogeneous systems) in 8 (or 10 on heterogeneous systems) of the 12 realistic applications in the experiment. Overall, the analysis in this chapter suggests that the algorithms proposed in this chapter have characteristics that outperform transformation based approaches, which may not be a good strategy to use particularly when the value of the sum of the minimum repetition number of actors is relatively high. The CA-STE scheduling algorithms proposed in this chapter are used as **Step 3** of the proposed scheduling framework in Section 1.5.

Chapter 5

Code-Size-Aware Mapping for SDFGs

5.1 Overview

Traditionally, code size reduction has been an important issue with embedded system design. Although the capacity of storage devices is continuously increasing and the unit costs are accordingly decreasing, the demand for storage space increases even faster [BFG+03]. In embedded systems, the cost of on-chip instruction memory, directly related to code size, is often comparable to the cost of the processing cores. In general, accessing on-chip memory is much faster than off-chip memory but at the same time also more expensive. Thus, as cost restrictions typically limit the size of on-chip memory, there is a strong interest in the development of code size reduction technologies [BFG+03]. Besides cost design considerations, embedded systems are often required to satisfy hard real-time constraints. This means that runtime predictability also becomes an issue. Overall, smaller code size can lead to fewer accesses of off-chip memory or even help avoid off-chip memory access altogether, which can improve the performance and power consumption of a system accordingly [DCdM+99].

In recent years, there has been lots of research related to scheduling and executing SDFGs on multicores [SGB06a, TBG+17a]. In brief, scheduling an SDFG on a multicore system consists of *mapping* actors onto different cores and deciding the execution of sequences of the actors on different cores. Mapping can be *actor-to-core binding based*, which means an actor can only be mapped on one core, or *actor duplication-enabled*, which means an actor can be mapped onto one or more cores [TBG+17a]. With binding based mapping, an actor is restricted to fire on a fixed processing core. In contrast, duplication-enabled mapping enables an actor to be fired on more than one processing core, making the parallelism within an application to be

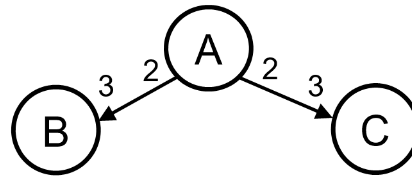


Figure 5.1: An example SDFG.

better utilized. Especially with SDFGs, an actor in an SDFG always fires multiple times within an iteration. For example, actors A, B, and C in Figure 5.1 have to fire at least 3, 2 and 2 times within an iteration, respectively. Therefore, duplication-enabled mapping is preferable in terms of improving throughput. However, the code size of the application is very likely to increase by actor duplication as an actor has to be mapped onto multiple cores. In addition, the commonly used unfolding technique for SDFG scheduling also brings about the actor duplication, which can lead to the increase of code size. The duplication of actors on multiple cores raises an important problem, which is the motivation for this chapter: code size reduction approaches for duplication-enabled mapping.

In this chapter, an ILP based exact mapping approach is proposed for duplication-enabled mapping on multicore systems, which aims at minimizing the extra code size introduced by the duplicated actors while maintaining the throughput obtained by the original scheduling algorithm. Since ILP based approaches are always time consuming and only suitable for small scale problems, a fast and effective mapping heuristic is also proposed, which can achieve near-optimal code size as shown in the experiment. Furthermore, this heuristic is jointly used with a CA-STE scheduling algorithm MERAS proposed in Chapter 4 to explore the trade-off between code size and throughput for the scheduling on systems considering the inter-core communication overhead.

The remainder of the chapter is organized as follows. Firstly, the problem definition is given in Section 5.2. In Section 5.3, a motivational example is used to illustrate the basic idea of this chapter. After that, the proposed algorithms are elaborated in Section 5.4, followed by an experimental evaluation in Section 5.5. Then, a case study is given to show how the proposed memory-aware scheduling framework schedules a realistic application on a homogeneous multicore system. Finally, Section 5.7 concludes this chapter.

5.2 Problem Definition

The problem addressed in this chapter is to propose a code-size-aware duplication-enabled scheduling strategy for an SDFG on a multicore system. The objective of the scheduling strategy is to reduce the extra code size caused by the duplication of actors without affecting the throughput of the original schedule. Before we extend the proposed mapping algorithms to communication-aware scheduling in Section 5.4.3, the target multicore processor is assumed to have a set of homogeneous processing cores, which are connected by an ideal interconnection network: this means that communication overheads between cores are assumed to be zero. If the communication overhead is ignored, for a given schedule with fixed starting time for each actor, the mapping of actors to cores does not change the starting time of these actors. Therefore, the throughput of an SDFG is not affected by its mapping results.

For the computation of the code size of a system, we make the following assumption. If an actor is mapped onto a core, the code of the actor is stored in the private memory of the core and cannot be shared with other cores. This means that if an actor is mapped on multiple cores, then multiple copies of the code of the actor should be stored separately on the corresponding cores. All code is stored in on-chip memory, and therefore no off-chip accessing is considered. Based on this assumption, code size in this chapter is calculated by the following equation:

$$CodeSize = \sum_{c}^{c \in C} \sum_{\alpha}^{\alpha \in MP(c)} CS(\alpha), \tag{5.1}$$

where C is a set which consists of all cores in the given multicore processor, $MP(c)$ is a set of actors which are mapped on core c and $CS(\alpha)$ is the code size of the actor α.

5.3 Motivational Example

The SDFG in Figure 5.1 is adopted as a motivational example. The numbers on the two ends of a channel are data producing and consuming rates of the connected actors. The firing time and code size of actors are all assumed to be 1. The SDFG is assumed to be mapped on a two-core system. For simplicity, retiming and unfolding are not considered during the scheduling of the example graph. Three different mapping strategies are applied to this example SDFG to illustrate the motivation of this chapter.

The outcome of the three schedules in Figure 5.2 is shown in Table 5.1. The columns

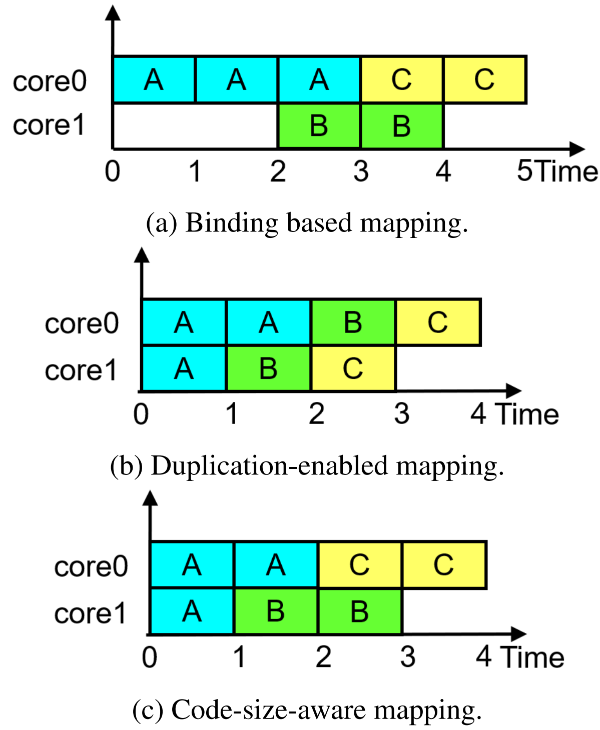


Figure 5.2: Three schedules of the example SDFG based on three mapping strategies respectively.

Table 5.1: Code size and IP got by three mapping methods.

	*core*0	*core*1	code size	IP
Duplication-enabled	{A, B, C}	{A, B, C}	6	4
Code-size-aware	{A, C}	{A, B}	4	4
Binding based	{A, C}	{B}	3	5

*core*0 and *core*1 give the actor distribution on the two processing cores. The column *code size* gives the total code size of actors on all cores. The IPs are shown in the column *IP*. As shown in the table, for the duplication-enabled mapping, its IP is the minimum among the three methods. However, the code size is much larger than the other two methods since code size reduction is not considered during its mapping process. For the binding based method, the code size is the minimum in these three methods since this method binds an actor onto one and only one processing core. However, throughput is accordingly restricted by this feature. One of the fastest schedules for the binding based mapping is shown in Figure 5.2a. The IP of the schedule is 5, which is longer than the schedules for the other two duplication-enabled mapping methods shown in Figure 5.2b and 5.2c. The reason is that the parallelism in the example graph is not

Table 5.2: Variables for the ILP Model.

Given Variables	
N	The number of actors
H	The number of instances
M	The number of cores
IP	The iteration period of a given schedule
T_i	The starting time of instance i
E_i	The execution time of instance i
CT_i	The source actor of instance i
CS_i	Code size of actor i
Variables to be Solved	
X_{ij}	Instance i is mapped on core j (binary variables)
U_{ij}	Instance i and instance j are mapped on the same core (binary variables)
A_{ij}	The number of instances of actor i on core j
CSC_{ij}	The code size of actor i on core j
$TOTAL$	The total code size of actors on all cores

fully utilized, as actor A is bound to core0 by the binding based mapping and the parallel execution of two entities of actor A on the two cores in Figure 5.2a and 5.2b is not allowed. The code-size-aware mapping method tries to decrease the code size of duplication-enabled scheduling under the constraint of maintaining the IP of the original schedule. As shown in Table 5.1, compared with the duplication-enabled method, the code-size-aware mapping gets same throughput and 33% less code size.

In this example, we can see that compared with binding based mapping, actor duplication can decrease the IP of the execution of an SDFG, i.e. increase the throughput of the system. The code size increase brought by the actor duplication can be effectively reduced if code-size-aware mapping is applied.

5.4 Algorithms

5.4.1 ILP Based Mapping Approach

Given a static schedule which consists of the starting time of each firing of each actor, an actor-to-core mapping with minimal code size can be obtained by ILP. We refer one firing of an actor as an *instance* of the actor. An actor is denoted as the source actor of

its instances. The variables used in the ILP model are given in Table 5.2. All variables to be solved are non-negative integers. The ILP model is formulated as follows.
Minimize

$$TOTAL = \sum_{i=1}^{N} \sum_{j=1}^{M} CSC_{ij} \tag{5.2}$$

Subject to

$$\begin{cases} \sum_{j=1}^{M} X_{ij} = 1, (i = 1, 2, ..., H) & (5.3) \\ X_{ik} + X_{jk} - Y_{ijk} \leq 1, (i, j = 1, 2, ..., H; k = 1, 2, ..., M) & (5.4) \\ X_{ik} + X_{jk} - 2 * Y_{ijk} \geq 0, (i, j = 1, 2, ..., H; k = 1, 2, ..., M) & (5.5) \\ U_{ij} = \sum_{k=1}^{M} Y_{ijk}, (i, j = 1, 2, ..., H) & (5.6) \\ IP * (U_{ij} - 1) + (T_i + E_i - T_j) \leq IP * (1 - Z_{ij}), (i, j = 1, 2, ..., H) & (5.7) \\ IP * (U_{ij} - 1) + (T_j + E_j - T_i) \leq IP * Z_{ij}, (i, j = 1, 2, ..., H) & (5.8) \\ A_{ij} = \sum_{k=1 | CT_k = i}^{H} X_{kj}, (i = 1, 2, ..., N; j = 1, 2, ..., M) & (5.9) \\ A_{ij} * CS_i \geq CSC_{ij}, (i = 1, 2, ..., N; j = 1, 2, ..., M) & (5.10) \\ A_{ij} * CS_i \leq CSC_{ij} * N * CS_i, (i = 1, 2, ..., N; j = 1, 2, ..., M) & (5.11) \\ CSC_{ij} = CS_i * B_{ij}, (i = 1, 2, ..., N; j = 1, 2, ..., M) & (5.12) \\ TOTAL \geq \sum_{i=1}^{N} \sum_{j=1}^{M} CSC_{ij} & (5.13) \end{cases}$$

The objective equation of the proposed ILP model is given in Equation 5.2. The first constraint, Equation 5.3, simply guarantees an instance can only be mapped on one core. Variable U_{ij} can be obtained using Equation 5.14.

$$U_{ij} = \sum_{k=1}^{M} X_{ik} * X_{jk} \tag{5.14}$$

However, Equation 5.14 is a nonlinear equation. Therefore, we introduce temporary binary variables Y_{ijk} to linearize Equation 5.14 as shown in Equations 5.4, 5.5 and 5.6.

Equations 5.4 and 5.5 make Y_{ijk} satisfy

$$Y_{ijk} = \begin{cases} 1, & if \;\; X_{ik} = 1, \wedge X_{jk} = 1; \\ 0, & if \;\; X_{ik} = 0, \vee X_{jk} = 0. \end{cases}$$

Then, Equations 5.7 and 5.8 are used to make sure that the instances with overlapped time slots cannot be mapped on the same core. Variable Z_{ij} is an auxiliary binary variable. When U_{ij} is 0, which means instances i and j are not mapped on the same core, Equations 5.7 and 5.8 are simplified as follows.

$$\begin{cases} -IP + (T_i + E_i - T_j) \leq IP * (1 - Z_{ij}) \\ -IP + (T_j + E_j - T_i) \leq IP * Z_{ij} \end{cases}$$

These two inequalities are always satisfied since the left-hand sides of the equations are always non-positive and the right-hand sides are always non-negative.

When $U_{ij} = 1$, Equations 5.7 and 5.8 become as follows:

$$\begin{cases} (T_i + E_i - T_j) \leq IP * (1 - Z_{ij}) & (5.15) \\ (T_j + E_j - T_i) \leq IP * Z_{ij} & (5.16) \end{cases}$$

Assume the time slots of instances i and j are overlapped, then Equations 5.17 and 5.18 must be held.

$$\begin{cases} T_i + E_i - T_j > 0 & (5.17) \\ T_j + E_j - T_i > 0 & (5.18) \end{cases}$$

Since the right-hand side of one of the Equations 5.15 and 5.16 must be 0, these two equations cannot be satisfied simultaneously given Equations 5.17 and 5.18. Therefore, the assumption is false, which means the time slots of instance i and j are not overlapped if $U_{ij} = 1$.

The variable A_{ij} which denotes the number of instances of actor i on core j is obtained by Equation 5.9. Then, the variable CSC_{ij} which denotes the code size of actor i on core j is achieved by Equations 5.10, 5.11 and 5.12. These three equations together make $CSCS_{ij}$ satisfy the condition

$$CSC_{ij} = \begin{cases} CS_i, & if \;\; A_{ij} > 0; \\ 0, & if \;\; A_{ij} = 0. \end{cases}$$

In Equation 5.12, B_{ij} is an auxiliary binary variable which constrains the value of CSC_{ij} to be either CS_{ij} or 0. For an actor with $CS_{ij} > 0$, when $A_{ij} = 0$, the value of CSC_{ij} is constrained to be 0 by Equation 5.10 since all variables to be solved are non-negative integers. Equation 5.11 is always satisfied in this case. When $A_{ij} > 0$, the value of CSC_{ij} has to be larger than 0 to satisfy Equation 5.10, which means CSC_{ij} can only be CS_{ij}. Meanwhile, Equation 5.10 is always true as the maximum value of CSC_{ij} is restricted to be CS_{ij} by Equation 5.12.

Note that for some dummy actors in an SDFG, the values of CS_i can be 0. Under these circumstances, the value of CSC_{ij} is restricted to be 0 by Equation 5.12 while Equations 5.10 and 5.11 are always satisfied. This is the reason why the value of CS_i is shown to be multiplied by both left side and right side of Equation 5.11. Otherwise Equation 5.11 can never be satisfied when $A_{ij} > 0$ and $CS_i = 0$.

Besides the basic constraints, it is always helpful to solve the ILP model to introduce some redundant constraints. We give some redundant constraints as follows.

$$
\begin{cases}
TOTAL \geq \sum_{i}^{N} CS_i & (5.19) \\
TOTAL \leq HB & (5.20) \\
\sum_{j}^{M} CSC_{ij} \geq CS_i & (5.21)
\end{cases}
$$

The HB in Equation 5.20 is the higher bound of the objective variable $TOTAL$, which can be obtained by the default mapping of the given schedule or a fast heuristic, e.g. the mapping heuristic proposed in this chapter.

5.4.2 Code-Size Aware Mapping Heuristic

Although the ILP based algorithm can guarantee an optimal result, the solving speed is slow and the runtime of the algorithm increases dramatically with the scale of the ILP model. Therefore, a lightweight yet efficient heuristic is also proposed in this chapter. The heuristic sequentially maps actors on cores in ascending order of the starting time of the actors. If more than one actor starts firing at the same time point, we use Algorithm 10 to break the tie. Then, Algorithm 11 is used to decide which core should an actor be mapped on. The input of the heuristic includes RA, a set of actors which are ready to fire; AC, a set of available cores; and $\boldsymbol{AL}$, a vector of current actor allocation on cores. We use c to denote an element in AC, and each c is a core number. $|AC|$ is the number

Algorithm 10 Select the actor with the highest priority α_p

Input:

The set of actors which are ready to fire *RA*;

Output:

The actor with the highest priority α_p

Iteration:

```
1: Set α_p to be the first actor in RA;
2: for all α ∈ RA do
3:   if NC(α_p) > NC(α) then
4:     α_p ← α;
5:   else if NC(α_p) = NC(α) then
6:     if ϒ(α_p)/CS(α_p) > ϒ(α)/CS(α) then
7:       α_p ← α;
8:     else if ϒ(α_p)/CS(α_p) = ϒ(α)/CS(α) then
9:       if ET(α_p) > ET(α) then
10:        α_p ← α;
11:      end if
12:    end if
13:  end if
14: end for
15: return α_p;
```

of available cores. An element $\boldsymbol{AL}(c)$ in $\boldsymbol{AL}$ is a set of actors which are allocated on core c; $|\boldsymbol{AL}(c)|$ is the number of elements in the set.

Select an Actor to be mapped

An algorithm to select the actor with the highest mapping priority, α_p, is shown in Algorithm 10. If an actor α is in the set $\boldsymbol{AL}(c)$, then core c is a *prior core* for the actor α, which means that at least one prior instantiation of actor α was allocated to core c. During the mapping process, two extra elements NC and Υ are introduced to the tuple of an actor α. For an actor α, $NC(\alpha)$ is the number of its prior cores in AC, and element $\Upsilon(\alpha)$ is the number of repetitions of α in one iteration of the SDFG. In particular, if no prior core exists for an actor α, then there is no difference to map the actor to any of the available cores. Therefore, $NC(\alpha)$ should be set as the number of available cores $|AC|$.

The four elements NC, Υ, CS and ET within each actor are used as criteria to judge the priority of actors. The input of the algorithm is RA, and the output is the actor with the highest priority α_p. The actor α_p is initialized as the first actor in RA in line 1 of Algorithm 10. The for-loop between lines 2 and 14 is used to find out the actor with the

highest priority. As shown in line 3 and line 4 in the algorithm, an actor with a smaller $NC(\alpha)$ value has a higher priority. This is because an actor with a small NC number has only limited options to do the mapping if we want to map the actor to one of its prior cores. If its prior cores are occupied by other actors at the current point in time, the actor will have to be allocated onto a non-prior core, which will lead to an increase of the code size. Therefore, it is reasonable to map the actor with the smallest NC first.

If the NC of two actors is equal, then the quotients of the repetition number $\boldsymbol{\Upsilon}$ and code size CS of actors are used as a second criterion, as shown in line 6 and line 7 in the algorithm. An actor with a larger repetition number has a lower priority because this actor is more likely to be distributed on multiple cores. This means the duplication of the code of this actor on various cores is less avoidable. On the other hand, since the duplication of an actor with a larger code size leads to a greater increase of code size than the duplication of an actor with a smaller code size, the actor with a larger code size should have a higher priority during the mapping process. Therefore, we jointly use the two factors $\boldsymbol{\Upsilon}$ and CS to decide the priority within two actors.

Then, if the quotients of $\boldsymbol{\Upsilon}$ and CS of two actors are also the same, the firing time ET is used as the final tie-breaker, as shown in line 9 to line 11 in the algorithm. An actor with a larger firing time has a lower priority. Since an actor with a larger ET value can occupy a core for a longer period, it is better to fire an actor with a smaller ET value first in order to regain the availability of the core sooner.

Map an Actor onto a Core

After α_p has been obtained, Algorithm 11 is used to get the core with the highest priority for α_p. The input of the algorithm consists of α_p, AC, and $\boldsymbol{AL}$. c_p is initialized to the first core in the set of available cores AC. The for-loop is used to get the core with the highest priority c_p for α_p. Mapping actor α_p onto one of its prior cores can reduce the overall code size by preventing the code duplication of the actor on a non-prior core. Therefore, the algorithm gives priority to prior cores of actor α_p as shown in line 7. If α_p has more than one prior core or it does not have any prior core, which means the condition in line 3 of the algorithm is met, then the value of $|\boldsymbol{AL}(c)|$ is used to break the tie. As shown in line 4 to line 6 of the algorithm, a core with a larger $|\boldsymbol{AL}(c)|$ has lower priority. A core with a larger $|\boldsymbol{AL}(c)|$ value means the core is the prior core for a larger number of actors. Therefore, the algorithm gives lower priority for this core and tries to reserve the core for other actors.

Algorithm 11 Get the core with the highest priority for α_p

Input:

The actor with the highest priority α_p, the set of available cores (AC), and the vector of current actor allocation on these cores ($\boldsymbol{AL}$)

Output:

The core with the highest priority c_p

Iteration:

1: Set c_p to be the first core in AC;
2: **for all** $c \in AC$ **do**
3:   **if** $(\alpha_p \in \boldsymbol{AL}(c)$ && $\alpha_p \in \boldsymbol{AL}(c_p))$ || $(\alpha_p \notin \boldsymbol{AL}(c)$ && $\alpha_p \notin \boldsymbol{AL}(c_p))$ **then**
4:     **if** $|\boldsymbol{AL}(c_p)| > |\boldsymbol{AL}(c)|$ **then**
5:       $c_p \leftarrow c$;
6:     **end if**
7:   **else if** $\alpha_p \in \boldsymbol{AL}(c)$ && $\alpha_p \notin \boldsymbol{AL}(c_p)$ **then**
8:     $c_p \leftarrow c$;
9:   **end if**
10: **end for**
11: **return** c_p

5.4.3 Extension for Communication-Aware Scheduling

So far in this chapter, inter-core communication time is not considered in the proposed code-size-aware mapping algorithm. If inter-core communication time is considered, both the throughput and code size of an application will be affected by its mapping on a multicore system. In this section, we extend one representative CA-STE based scheduling algorithm proposed in Chapter 4, MERAS, to explore the trade-off between throughput and code size of a multicore system. Here, we introduce a code size factor F_{cs} to control the preference of the mapping for code size and throughput. The proposed code-size-aware mapping algorithm for MERAS is shown in Algorithm 12. This mapping algorithm checks all pairs of ready actors and processing cores. As shown in lines 7 and 19, the prior actor-core pair (α_p, c_p) is updated directly as an actor-core pair (α, c) only if the ready time of the actor-core pair (α, c) is $(F_{cs} * LB)$ time lower than the ready time of the prior pair (α_p, c_p). Otherwise, if the ready time of the actor-core pair (α, c) is not $(F_{cs} * LB)$ higher than the ready time of the prior pair (α_p, c_p), the code size optimization conditions are used as shown in lines 9-17 and lines 21-33. Lines 10-16 and lines 22-32 in Algorithm 12 are similar to lines 3-9 in Algorithm 11 and lines 3-13 in Algorithm 10, respectively.

After a periodic schedule S has been obtained by MERAS, a code-size-aware rescheduling process for one period of the schedule S is conducted to optimize the code

Algorithm 12 Code-size-aware mapping for MERAS

Input:

The set of actors which are ready to fire RA; the set of all cores AC; and the code size factor F_{cs};

Output:

The actor-core pair with the highest priority (α_p, c_p)

Iteration:

1: Set α_p to be the first actor in RA and c_p to be the first core in AC;
2: Let $t_{er} = +\infty$;
3: **for all** $\alpha \in RA$ **do**
4: Let $t'_{er} = +\infty$;
5: **for all** $c \in AC$ **do**
6: Get the earliest ready time t_t of actor α on core c using Algorithm 8;
7: **if** $t'_{er} - t_t > F_{cs} * LB$ **then**
8: $c'_p \leftarrow c$; $t'_{er} \leftarrow t_t$;
9: **else if** $t'_{er} - t_t > -F_{cs} * LB$ **then**
10: **if** $(\alpha_p \in \boldsymbol{AL}(c)$ && $\alpha_p \in \boldsymbol{AL}(c_p))$ || $(\alpha_p \notin \boldsymbol{AL}(c)$ && $\alpha_p \notin \boldsymbol{AL}(c_p))$ **then**
11: **if** $|\boldsymbol{AL}(c_p)| > |\boldsymbol{AL}(c)|$ **then**
12: $c'_p \leftarrow c$; $t'_{er} \leftarrow t_t$;
13: **end if**
14: **else if** $\alpha_p \in \boldsymbol{AL}(c)$ && $\alpha_p \notin \boldsymbol{AL}(c_p)$ **then**
15: $c'_p \leftarrow c$; $t'_{er} \leftarrow t_t$;
16: **end if**
17: **end if**
18: **end for**
19: **if** $t_{er} - t'_{er} > F_{cs} * LB$ **then**
20: $\alpha_p \leftarrow \alpha$; $c_p \leftarrow c'_p$; $t_{er} \leftarrow t'_{er}$;
21: **else if** $t_{er} - t'_{er} > -F_{cs} * LB$ **then**
22: **if** $NC(\alpha_p) > NC(\alpha)$ **then**
23: $\alpha_p \leftarrow \alpha$; $c_p \leftarrow c'_p$; $t_{er} \leftarrow t'_{er}$;
24: **else if** $NC(\alpha_p) = NC(\alpha)$ **then**
25: **if** $\boldsymbol{\Upsilon}(\alpha_p)/CS(\alpha_p) > \boldsymbol{\Upsilon}(\alpha)/CS(\alpha)$ **then**
26: $\alpha_p \leftarrow \alpha$; $c_p \leftarrow c'_p$; $t_{er} \leftarrow t'_{er}$;
27: **else if** $\boldsymbol{\Upsilon}(\alpha_p)/CS(\alpha_p) = \boldsymbol{\Upsilon}(\alpha)/CS(\alpha)$ **then**
28: **if** $ET(\alpha_p) > ET(\alpha)$ **then**
29: $\alpha_p \leftarrow \alpha$; $c_p \leftarrow c'_p$; $t_{er} \leftarrow t'_{er}$;
30: **end if**
31: **end if**
32: **end if**
33: **end if**
34: **end for**
35: **return** (α_p, c_p);

size. The code-size-aware rescheduling process is the same as the scheduling process of MERAS except that the code-size-aware mapping heuristic in Algorithm 12 is used as the mapping strategy instead of the actor-to-core allocation rule of MERAS in Chapter 4. The rescheduling process stops when the execution of all the actors in one period of the schedule S has ended. The schedule obtained by the rescheduling process is used as the new code-size-aware periodic schedule. The scheduling algorithm which combines the communication-aware scheduling MERAS and the code-size-aware rescheduling for MERAS is denoted as CCSAS (Communication-and-Code-Size-Aware Scheduling) in this chapter. The design space between throughput and code size can be explored by changing the code size factor F_{cs}.

5.5 Evaluation

5.5.1 Experimental Setup

Two datasets are used in this experiment. The first dataset consists of 16 realistic applications which are all obtained from the benchmark folder of Streamit 2.1.1 (Older release) [Str18]. Specifically, the 16 applications include BeamFormer (denoted as BeamFormer1) and SerializedBeamFormer under the subfolder beamformer; FFT2, FFT3 and FF4 under the subfolder fft; FilterBankNew under the subfolder filterbank; FMRadio under the subfolder fm; MatrixMultBlock under the subfolder matmul-block; and BeamFormer (denoted as BeamFormer2), BitonicSort, ChannelVocoder, FFT5, FMRadio5, MPEGdecoder, tde_pp and VocoderTopLevel under the subfolder asplos06.

Another dataset is also used to enhance the evaluation and understanding of the proposed mapping heuristic, which consists of 5 groups of random SDFGs generated by a tool named SDF3 [SGB06b]. The 5 SDFG groups are denoted as *Random10*, *Random20*, *Random30*, *Random40* and *Random50*. Each of these groups includes 100 different SDFGs with 10, 20, 30, 40 or 50 actors, respectively. In the five groups, the sum of the elements in the repetition vector of an SDFG is set to be 30, 60, 90, 120, 150, respectively. Other generation parameters for the five groups are the same. The number of the channels connected to an actor is generated randomly with an average of 3, variance of 1, minimum of 1 and maximum of 5. The data producing and consuming rates of actors are generated randomly with an average of 3, variance of 9, minimum of 1 and maximum of 30. The execution time of actors is generated randomly with an average of 50, variance of 250, minimum of 10 and maximum of 100. The probability

Table 5.3: The code size obtained by DES, CSAS and CSMS and the runtime of CSAS and CSMS for 8 realistic applications when $M = 2$.

Application	Code Size (Byte)			Runtime	
	DES	CSAS	CSMS	CSAS (ms)	CSMS (s)
BeamFormer1	19,267	19,267	19,267	173.07	814.87
BitonicSort	1,238	1,238	1,238	26.63	43.86
FMRadio5	68,786	42,437	37,213	35.79	187.38
BeamFormer2	24,744	24,744	24,744	155.09	39.52
SerializedBeamFormer	24,744	24,744	24,744	111.99	58.87
FFT2	6,588	6,588	6,480	44.56	2,554.64
FFT5	5,539	5,539	5,539	121.14	53.92
FMRadio	57,142	57,142	57,142	244.41	1,520.57

that initial tokens exist on a channel is set to 0. The code size of each actor in these randomly generated SDFGs is taken from a uniformly distributed random distribution with values between 10 and 100.

In the experiments, the schedules obtained by the duplication-enabled scheduling algorithm $Sch_p(G,P)$ from [ZGBS16] are used as the input for the proposed mapping algorithms. Since the actor-to-core mapping of $Sch_p(G,P)$ is not explicitly given in [ZGBS16], we assume that an actor is allocated on the first available core in the core list. We denote this mapping as DES. For the proposed mapping algorithms, we denote the ILP based exact approach and the heuristic as CSMS and CSAS , respectively. Lingo 17.0 [Sys18] is used as the ILP solver for CSMS. A runtime limit of one hour is applied to CSMS.

For the communication-aware mapping algorithm CCSAS, the applications are assumed to be scheduled on a bus-based homogeneous multicore processor with 8 cores running at 500 MHz while the bandwidth of the bus is 8 GB/s, the same as the processor used in the experiment of Chapter 4.

The auto-concurrency degree of all the SDFGs used in the experiment is restricted to 1 by default, which means multiple simultaneous firings of an actor are not allowed. The restriction can be applied to an SDFG by adding a self-edge with one initial token to every actor in the SDFG.

5.5.2 Realistic Applications

For CSMS, we only examine the situation where the number of cores is 2, as the runtime of CSMS increases exponentially with the number of cores. CSMS can finish within the

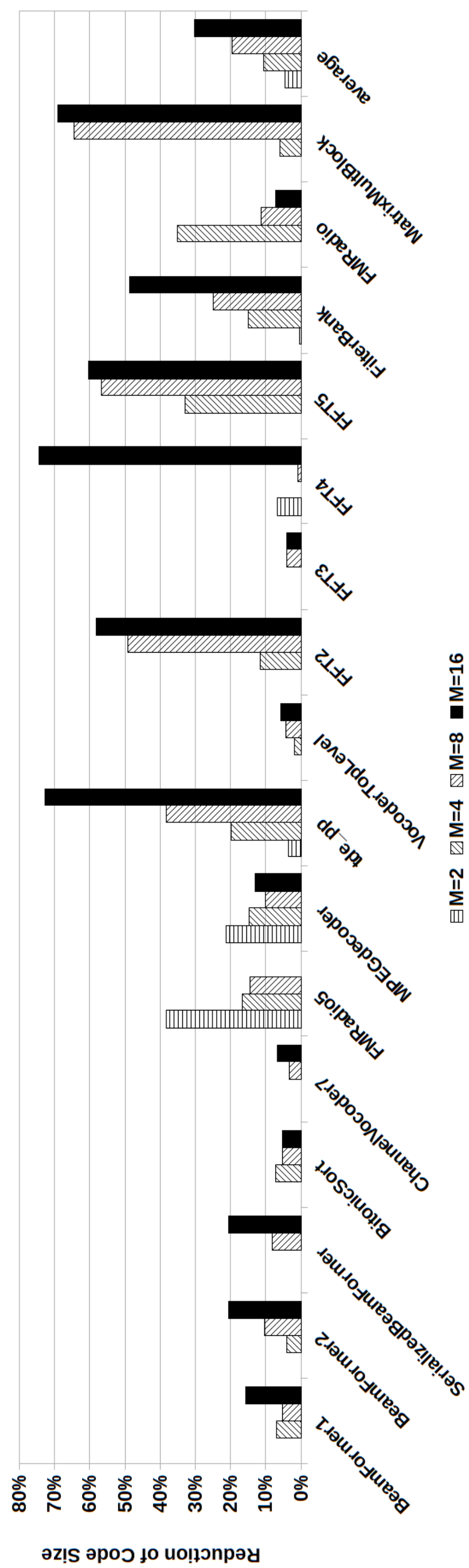


Figure 5.3: Code size reduction of CSAS for realistic SDFGs on multicore processors with 2, 4, 8 and 16 cores compared with DES.

time limit for 8 of the 16 realistic applications listed in Table 5.3, where the code size obtained by DES, CSAS and CSMS and the runtime of CSAS and CSMS are given. As shown in Table 5.3, CSMS gets a significant code size reduction (45.9%) than DES for FMRadio5. The proposed heuristic CSAS is also able to achieve 38.3% less code size than DES for FMRadio5, which is 14.0% larger than the minimum code size obtained by CSMS. For FFT2, a slight code size reduction (about 1.6%) can be obtained by CSMS compared to DES. For the other six of the eight applications, the minimum code size can be obtained by DES directly when the number of cores is 2. The runtime of CSMS can be orders of magnitude higher than the runtime of CSAS for the eight applications in Table 5.3.

Compared with DES, the code size reduction of CSAS for different SDFGs on processors with different number of cores is shown in Figure 5.3, where M is the number of cores. The reduction is calculated by $(CodeSize_d - CodeSize_c)/CodeSize_d$, where $CodeSize_d$ is the code size obtained by DES and $CodeSize_c$ is the code size obtained by CSAS. The throughput obtained by CSAS for all the realistic SDFGs is the same as DES. From Figure 5.3, we can see that CSAS achieves up to 75% code reduction compared with DES. The average code size reduction of all the tested SDFGs is also given in Figure 5.3, labeled *average* (right-end of the figure). The average code size reduction on a two-core system is about 4%, while the average code size reduction on the processor with 16 cores is about 30%. The average code size reduction of SDFGs increases with the number of cores. A similar trend can also be observed with most of the SDFGs in Figure 5.3. This suggests that CSAS performs better when the computing resources are abundant.

The reason for this can be explained easily. When the number of cores in a multicore system is very small, the parallelism in an SDFG cannot be sufficiently utilized by the system. In this case, the duplication of actors is not necessary and therefore the overall code size can be small anyway. As the number of the cores in the system increases, the duplication of actors starts to improve the throughput of the system, but the probability that an actor is not mapped onto a prior core also increases thereby increasing overall code size. At the same time, this gives more scope and potential for CSAS, which takes into account core locality for actors, to achieve a high code size reduction.

5.5.3 Randomly Generated SDFGs

For the second dataset, consisting of five groups of randomly generated SDFGs, the average code size obtained by CSAS and DES for each of the five SDFG groups is shown

Table 5.4: The code size obtained by CSAS and DES for random SDFGs.

#cores / #name	2			4			8			16			32			64		
	CSAS	DES	Red.	CSAS	DES	Red.	CSAS	DES	Red.	CSAS	DES	Red.	CSAS	DES	Red.	CSAS	DES	Red.
Random10	1.12	1.21	7%	1.08	1.32	18%	1.03	1.31	20%	1.00	1.31	22%	1.00	1.31	22%	1.00	1.31	22%
Random20	1.09	1.12	2%	1.06	1.19	10%	1.02	1.20	14%	1.01	1.20	15%	1.00	1.20	16%	1.00	1.20	16%
Random30	1.07	1.09	2%	1.09	1.21	9%	1.02	1.18	13%	1.01	1.18	14%	1.00	1.18	15%	1.00	1.18	15%
Random40	1.09	1.10	1%	1.07	1.17	8%	1.03	1.18	12%	1.02	1.18	13%	1.01	1.18	13%	1.00	1.18	14%
Random50	1.06	1.09	2%	1.06	1.13	5%	1.02	1.14	10%	1.01	1.14	11%	1.00	1.14	12%	1.00	1.14	12%

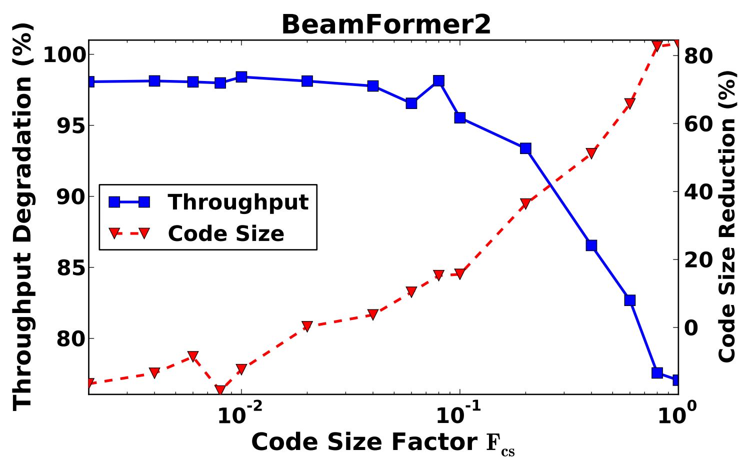
in Table 5.4. The top row of the table shows the number of cores considered in each case. The leftmost column of the table gives the names of the five SDFG groups. The throughput obtained by CSAS and DES is not given in the table since they both get the same throughput for all the SDFGs tested in the experiment.

The code sizes in Table 5.4 are all normalized by the minimum code sizes of the corresponding SDFGs. The minimum code size of an SDFG is the sum of the code size of each actor. Thus, if the obtained code size for an SDFG is 1, the obtained code size equals to the minimum code size of the SDFG. The code sizes obtained by CSAS on a 64-core system for all the five groups are all 1, which suggests that CSAS achieves minimum code size for each SDFG in all the five groups when the number of cores is larger than the number of actors. The *Red.* column in Table 5.4 is the code size reduction of CSAS relative to DES. For all the five groups, the normalized code sizes obtained by CSAS are less than DES. The normalized code size obtained by CSAS decreases as the number of cores increases, while the normalized code size obtained by DES generally shows an opposite tendency. When the number of actors is much smaller than the number of cores, the code size reduction of CSAS compared with DES is not remarkable (as low as 1% for Random40 on a 2-core system as shown in Table 5.4). The code size reduction increases to 12%-22% for the five groups when the number of cores is large enough. In conclusion, the advantage of CSAS relative to DES observed in Section 5.5.2 is observed again in this section when using a large number of random SDFGs.

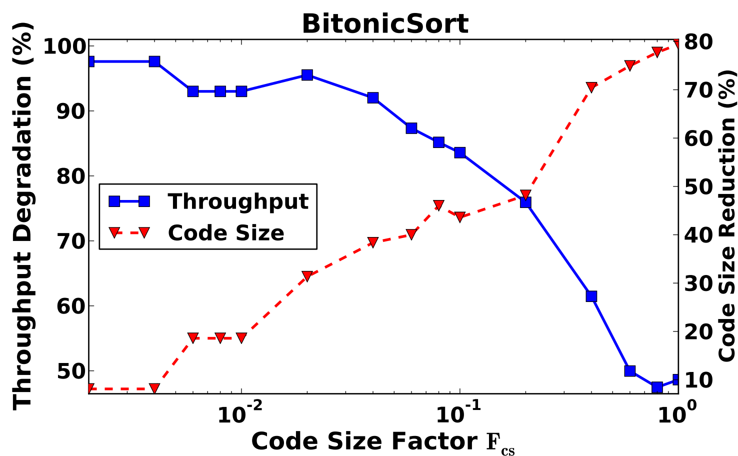
5.5.4 The Trade-Off between Throughput and Code Size for Communication-Aware Scheduling

In this section, we examine the trade-off between throughput and code size obtained by the proposed algorithm CCSAS. The eight realistic applications BeamFormer2, BitonicSort, ChannelVocoder, FFT5, FMRadio5, MPEGdecoder, tde_pp and VocoderTopLevel are adopted. The restriction for the auto-concurrency degree is removed to show the relationships between the throughput and code size obtained by CCSAS more clearly.

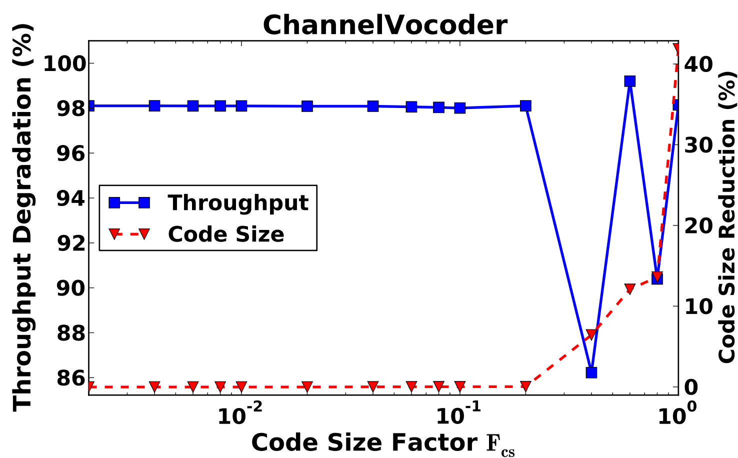
The experimental results are shown in Figure 5.4. The throughput degradation in Figure 5.4 is calculated by TH_c/TH_m, where TH_c and TH_m are the throughput obtained by CCSAS and MERAS, respectively. Similar to previous definition, the code size reduction in Figure 5.4 is obtained by $(CodeSize_m - CodeSize_c)/CodeSize_m$,



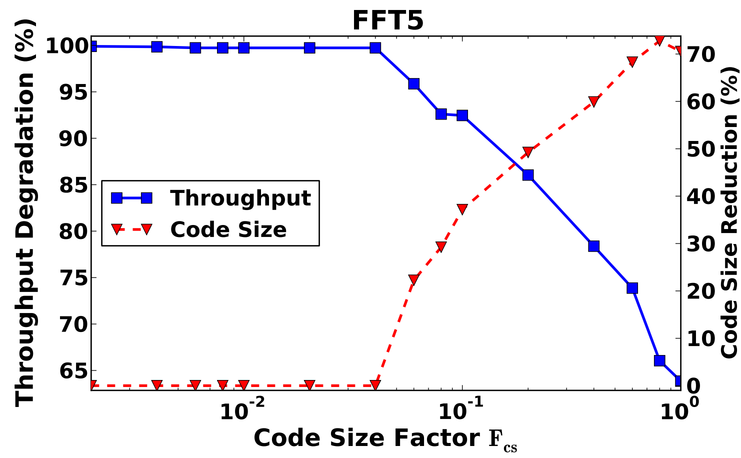
(a) BeamFormer2.



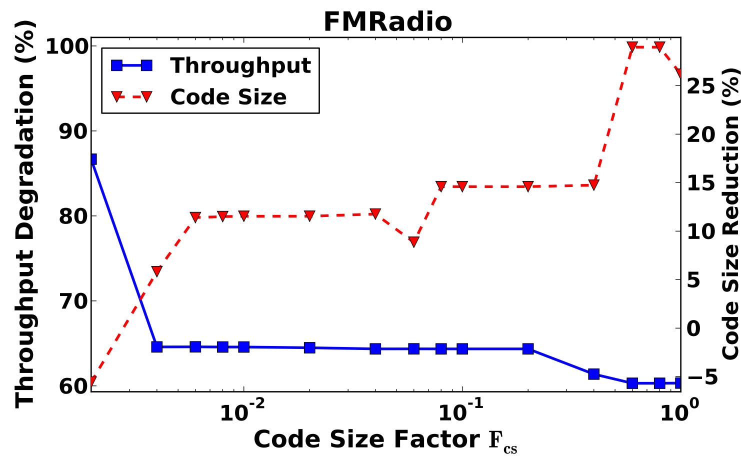
(b) BitonicSort.



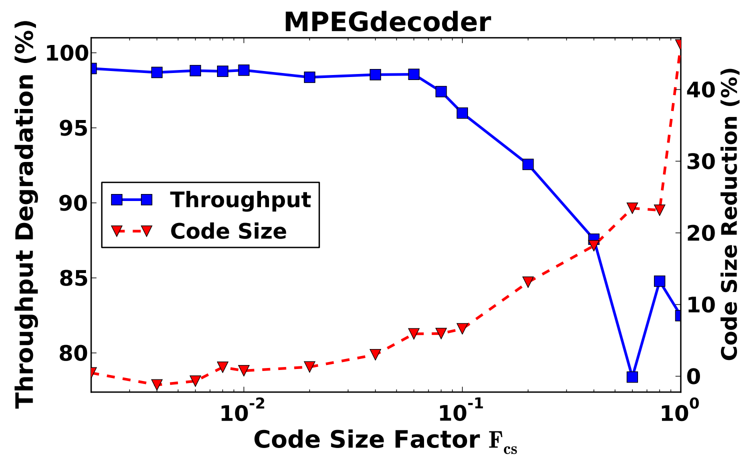
(c) ChannelVocoder.



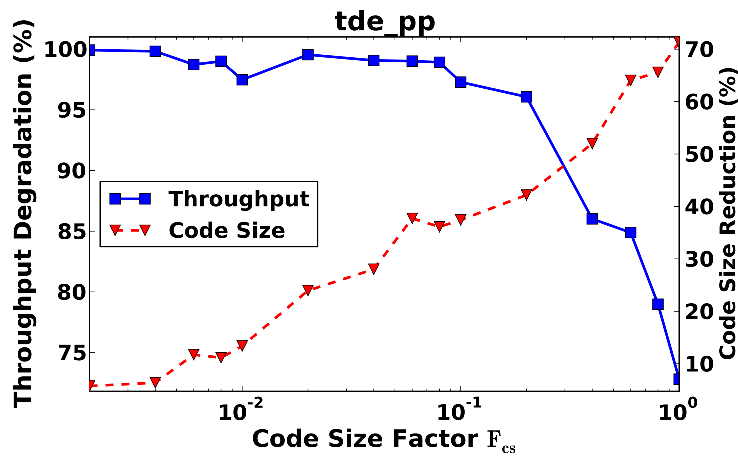
(d) FFT5.



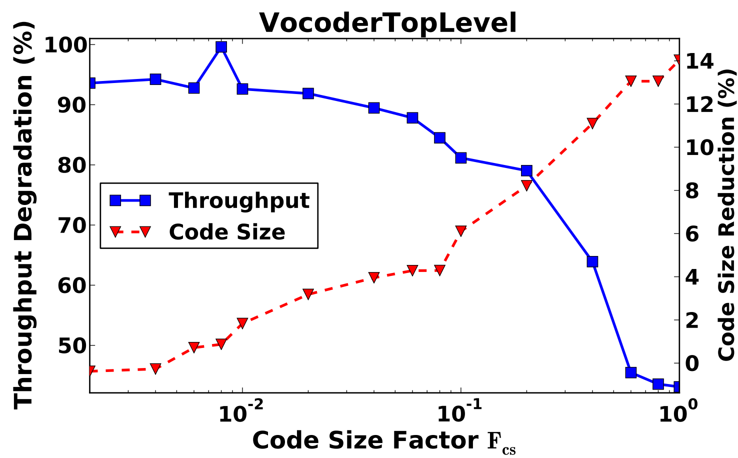
(e) FMRadio5.



(f) MPEGdecoder.



(g) tde_pp.



(h) VocoderTopLevel.

Figure 5.4: The trade-off between the throughput and code size obtained by CCSAS for eight realistic applications.

Table 5.5: The code size reduction obtained by CCSAS for eight applications when throughput degradation is 95%, 90% and 80%.

Application	Code Size Reduction		
	$D_{thr} = 95\%$	$D_{thr} = 90\%$	$D_{thr} = 80\%$
BeamFormer2	15.64%	36.32%	65.78%
BitonicSort	31.34%	38.36%	46.01%
ChannelVocoder	41.87%	41.87%	41.87%
FFT5	22.25%	37.16%	49.23%
FMRadio5	0.00%	0.00%	0.00%
MPEGdecoder	6.61%	13.12%	46.22%
tde_pp	42.18%	42.18%	64.05%
VocoderTopLevel	0.87%	3.18%	6.11%
Average	20.10%	26.53%	39.91%

where $CodeSize_c$ and $CodeSize_m$ indicate the code size obtained by CCSAS and MERAS, respectively. For all the eight applications, in general, the code size reduction of CCSAS increases as the code size factor F_{cs} increases while the throughput obtained by CCSAS decreases with the value of F_{cs}. However, a remarkable fact is that with the increase of F_{cs}, the decreasing degrees of throughput and code size are not the same. The code size reduction obtained by CCSAS for the eight applications is shown in Table 5.5, where D_{thr} is used to denote the throughput degradation of CCSAS for MERAS. From Table 5.5, we can observe that for five of the eight applications BeamFormer2, BitonicSort, ChannelVocoder, FFT5 and tde_pp, CCSAS can get 15.64%, 31.34%, 41.87%, 22.25%, and 42.18% of code size reduction respectively compared with MERAS while maintaining 95% of the throughput obtained by MERAS. When the throughput degradation for MERAS is reduced to 80%, six of the eight applications can get more than 40% of code size reduction by CCSAS relative to MERAS. On average, the code size reduction of CCSAS for the eight applications is 20.10%, 26.53% and 39.91% when the throughput obtained by CCSAS is 95%, 90% and 80% of the throughput obtained by MERAS, respectively.

5.6 A Case Study of the Proposed Memory-Aware Scheduling Framework

5.6.1 Setting

As the three key steps in the proposed memory-aware scheduling framework (see Section 1.5) have all been introduced (**Step 1** in Chapter 3, **Step 3** in Chapter 4 and

Table 5.6: The buffer usage obtained by NAIVE, ZHU and EXACT for BitonicSortRecursive.

	NAIVE	ZHU	EXACT
buffer usage (byte)	13,375	6,192	6,160

Step 4 in this chapter). In this section, a case study is presented to show how the proposed memory-aware scheduling framework combining the three steps works. A bitonic sorting application named BitonicSortRecursive is adopted, which is obtained from the folder streamit-src-2.1/apps/benchmarks/bitonic-sort of Streamit 2.1.1 (Older release) [Str18]. In the SDFG of BitonicSortRecursive, there are 397 actors and 453 channels, while the sum of the repetition number of actors is 538. BitonicSortRecursive is assumed to be scheduled on a bus-based multicore processor with eight homogeneous processing cores. Same with the experimental settings in Sections 4.5 and 5.5, the processor runs at 500 MHz and the bandwidth of the bus in the processor is 8 GB/s. The objective of the scheduling is to minimize memory usage under the constraint of 6 times' speedup compared with a single core processor. Here, the speedup is the same as previously defined in Section 4.5.2.

5.6.2 Scheduling

In **Step 1** of the proposed framework, the buffer usage of BitonicSortRecursive is minimized. The buffer usage obtained by three scheduling algorithms is shown in Table 5.6, where NAIVE is Algorithm 2 in [ZGBS12], ZHU is the buffer minimization algorithm proposed in [ZGBS14], and EXACT is the exact buffer minimization algorithm proposed in Chapter 3. The buffer usage obtained by NAIVE is used in [ZGBS14] as an upper bound of the buffer usage for ZHU. As shown in Table 5.6, ZHU and EXACT can both achieve more than 50% less buffer usage compared to NAIVE, while the buffer usage obtained by EXACT is 32 bytes less than the buffer usage obtained by ZHU.

The buffer usage obtained by **Step 1** is modelled as buffer constraints on channels in **Step 2** of the proposed framework. The buffer constraints on channels are represented by a set BC, where each element $BC(e)$ denotes the buffer size on a channel e. During an STE based scheduling process, the size of the tokens on a channel cannot exceed the buffer size of this channel.

Then, in **Step 3**, a communication-aware STE based algorithm is used to schedule the buffer-constrained SDFG of BitonicSortRecursive on the 8-core bus-based multicore processor. Here, MERAS proposed in Chapter 4 is adopted. Then, a schedule is obtained

Table 5.7: The code size and speedup obtained by CCSAS using different values for the code size factor F_{cs}.

F_{cs}	speedup	code size (byte)
0	6.50	22,870
0.002	6.47	20,864
0.004	6.43	19,528
0.006	6.39	20,492
0.008	6.35	18,382
0.01	6.27	17,862
0.02	6.18	15,648
0.04	5.97	13,832
0.06	5.62	12,574
0.08	5.48	12,620
0.1	5.34	11,478
0.2	4.75	10,504
0.4	3.55	9,808
0.6	3.30	9,816
0.8	3.19	9,628
1	3.16	9,532

with 22,870 bytes' code size and 6.5 times' speedup compared to a single core. Based on this schedule, in **Step 4**, a code-size-aware mapping algorithm CCSAS proposed in Chapter 5 is used to explore the trade-off between the speedup and code size of mappings. As shown in Table 5.7, with the increase of the code size factor F_{cs}, the code size obtained by CCSAS reduces rapidly while the speedup obtained by CCSAS decreases gradually. When $F_{cs} = 0.02$, 31.58% less code size (7,222 bytes) is gained by CCSAS with only 5.06% of the speedup sacrifice compared to the schedule obtained in **Step 3**. The speedup is 6.18, which satisfies the scheduling constraint.

After **Step 4** has finished, one iteration of the scheduling framework is finished. If the memory usage obtained by one iteration cannot satisfy the design requirement, another iteration of the framework could be conducted until the design requirement is met.

Summary

Through the case study, significant memory usage reduction can be achieved by the proposed scheduling framework. Since there is no existing mapping algorithm to reduce the code size caused by actor duplication, the proposed framework achieves significant code size reduction in **Step 4** by using the code-size-aware mapping algorithm CCSAS

proposed in Chapter 5. Compared to a state-of-the-art buffer minimization algorithm ZHU, the exact buffer minimization algorithm proposed in Chapter 3 also achieves 32 bytes of buffer usage reduction. The CA-STE based scheduling in **Step 3** can effectively support the buffer constraints obtained in **Step 1** and generate high-quality schedules for **Step 4**. In summary, buffer usage reduction, communication-aware scheduling and code-size-aware mapping are combined successfully in the proposed scheduling framework.

5.7 Summary

In this chapter, an ILP based exact approach and a fast heuristic are proposed to reduce the code duplication of actors on multicore systems during scheduling. The experimental results show that compared with a naive mapping strategy, the proposed ILP based approach and fast heuristic can always get lower code size for all the SDFGs in the experiment. The proposed heuristic can get near-optimal code size using extremely short runtime compared with the proposed ILP based approach. Furthermore, to support communication-aware scheduling, the proposed mapping heuristic is combined with MERAS in Chapter 4 to explore the trade-off between throughput and code size of a multicore system during scheduling. For most of the applications used in the experiment, the proposed mapping heuristic can achieve significant code size reduction with minor throughput degradation for communication-aware scheduling. The code-size-aware mapping algorithms proposed in this chapter are used as **Step 4** of the proposed scheduling framework in Chapter 1. Finally, a case study is given as an example of applying the proposed memory-aware scheduling framework to schedule a realistic application on a bus-based homogeneous multicore system.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, a memory-aware scheduling framework for SDFGs on multicore systems has been proposed. The buffer size, throughput and code size of SDFGs on multicore systems are optimized successively in three key steps of the proposed framework. These three steps are connected by two modelling steps which could be implemented using existing modelling techniques. Algorithms for the three key steps buffer minimization, communication-aware scheduling and code-size-aware mapping are elaborated in Chapter 3, Chapter 4 and Chapter 5 of this thesis, respectively.

In Chapter 3, the STE based scheduling is extended to eliminate the unnecessary buffer size introduced by the initial tokens. Based on the extended STE based scheduling and an existing searching algorithm proposed in [SGB06a], an exact searching algorithm is proposed to get the minimum buffer size of an SDFG under the constraint of rate-optimal scheduling. To accelerate the searching process of the proposed algorithm, a branch-and-bound based method is also proposed to get a tight lower bound of the searching process. In addition, the buffer size obtained by [ZGBS14] is used as an upper bound of the searching process. To handle large-scale problems and avoid long processing time, the proposed exact algorithm is further extended to a fast and effective heuristic in Chapter 3. The experiment shows that the proposed buffer minimization heuristic can get as high as 56.5% buffer size reduction compared with a state-of-the-art buffer minimization algorithm proposed in [ZGBS14].

In Chapter 4, the STE approach is further extended to support communication-aware scheduling of SDFGs on bus-based multicore systems. A more complicated state containing more variables is developed for the communication-aware STE process. The

actor execution rule is also examined in this chapter. Traditionally, the STE process requires an actor to execute as soon as it is ready to execute. This rule works well when communication overhead is not considered in the STE process. However, inspired by the list-based scheduling algorithms for DAGs, more effective actor execution rules may exist for the communication-aware STE process. Therefore, four actor-to-core allocation rules are proposed in Chapter 4 to construct four communication-aware scheduling algorithms for homogeneous multicore systems. Two of the proposed scheduling algorithms are further extended for heterogeneous systems. Twelve realistic applications constitute the benchmark used in the experiment of this chapter. Experimental results show that compared with two well-known scheduling algorithms DLS and HEFT, the proposed scheduling algorithms can achieve on average about 50% higher throughput when 16 homogeneous cores are used. For the three heterogeneous situations in the experiment, the speedup obtained by the proposed algorithm is on average 86%, 90%, and 86% higher than HDLS when the number of cores is 16, respectively.

In Chapter 5, code-size-aware mapping algorithms are proposed to reduce the duplication of the code of actors on multiple cores. Firstly, an ILP based approach is proposed to give an exact solution for the code size minimization problem. Then, a heuristic is also proposed to get near-optimal solutions efficiently. Lastly, the proposed mapping heuristic is jointly used with a communication-aware scheduling algorithm proposed in Chapter 4 to explore the trade-offs between throughput and code size of a multicore system during scheduling. The experiment shows that the proposed heuristic can get near-optimal results compared with the proposed ILP based exact approach. Compared with a simple mapping strategy, the proposed heuristic can get about 30% code size reduction on average when the number of processing cores is 16. When the proposed mapping heuristic is jointly used with a communication-aware scheduling algorithm, trade-offs between throughput and code size can be clearly observed for all the SDFGs used in the experiment. In particular, a remarkable reduction of the code size (20.10% on average) can be achieved by the proposed mapping heuristic with minor throughput degradation (less than 5%) for most of the SDFGs used in the experiment.

The scheduling algorithms proposed in this thesis all adopt STE based scheduling. Compared with other scheduling approaches, STE based approach can schedule SDFGs directly without any graph conversion process. The retiming and unfolding of SDFGs are completed automatically through the STE process. The success of extending STE to support communication-aware scheduling in Chapter 4 shows the capability of STE based scheduling for handling complicated scheduling models. In Chapter 5, the

proposed mapping heuristic can be easily embedded into a CA-STE algorithm proposed in Chapter 4, which shows the flexibility of STE based scheduling. Thus, STE based scheduling could be a promising approach to construct fast and effective scheduling heuristics for more complicated and realistic scheduling models.

To sum up, in the proposed scheduling framework, throughput, buffer size and code size are all improved for SDFGs on multicore systems considering communication overhead. Experimental results validate our claims that the proposed algorithms in the framework outperform the existing algorithms in buffer size, throughput and code size in most experiment cases. s

6.2 Future Work

The proposed memory-aware scheduling framework gives an initial structure for optimizing both throughput and memory use for SDFGs on multicore systems. The framework could be further refined in the following ways.

- A conservative assumption used in Chapter 3 is that the buffer of one channel cannot be shared with other channels. By removing this assumption and allowing buffer sharing between different channels mapped on the same memory, a lower buffer requirement could be obtained.

- For the communication-aware scheduling in Chapter 4, as shown in the experiments, the runtime of the proposed STE based scheduling is extremely long on a rare number of cases. This is because the number of STATEs is extremely large and two identical STATEs are hard to be found for these cases. Therefore, a problem to be solved is to develop efficient methods that restrict/reduce the number of STATEs and guarantee some high efficiency for communication-aware STE based scheduling algorithms.

- The work in Chapter 4 only aims at an arbitrated bus-based architecture. For other communication architectures, the effectiveness and runtime of the proposed CA-STE based scheduling could be affected by the complexity of the communication model. Therefore, whether the proposed CA-STE based scheduling could work properly on more complicated communication architectures should be investigated further.

- In **Step 5** of the proposed scheduling framework, the scheduling and mapping

decisions obtained by **Steps 1-4** should be modelled as extra constraints for the next iteration of the scheduling framework. However, as discussed in Section 2.6, existing modelling approaches can only handle HSDFGs or binding based scheduling. Therefore, generic modelling approaches for the scheduling and mapping of SDFGs are desirable for the proposed scheduling framework.

- Although memory use is optimized in the proposed framework, the improvement of the throughput obtained by the memory use reduction is not quantified in this thesis. Therefore, a quantitative analysis for the memory use and throughput on a multicore system should be done in future work.

- Except for throughput and memory, other important criteria for streaming applications on embedded systems should also be considered in future work, such as energy consumption, latency, fault-tolerance, etc. In addition, the scheduling of concurrently executing applications is also an interesting problem.

- The algorithms in this thesis are all implemented and tested in SDF3 [SGB06b], meaning the experimental results are all obtained by theoretical analysis and simulations. Thus, a meaningful topic of further research is to apply this framework on realistic hardware platforms to examine the effectiveness of the proposed framework in a more accurate way.

- Since SDFGs cannot model the dynamic behaviours of applications, not all streaming applications can be modelled by SDFGs. Therefore, extending the framework to support more complicated models (e.g., Cyclo-Static Dataflow [BELP96], Scenario-Aware Dataflow [TGB+06], etc.) is also desirable.

Bibliography

[ABEDK16] Emmanuel Agullo, Olivier Beaumont, Lionel Eyraud-Dubois, and Suraj Kumar. Are static schedules so bad? A case study on cholesky factorization. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 1021–1030. IEEE, 2016.

[Adé96] Marleen Adé. *Data memory minimisation for synchronous data flow graphs emulated on DSP-FPGA targets*. PhD thesis, 1996.

[AHSvdP15] Waheed Ahmad, Philip KF Hölzenspies, Mariëlle Stoelinga, and Jaco van de Pol. Green computing: Power optimisation of vfi-based real-time multiprocessor dataflow applications. In *Proceedings of the Euromicro Conference on Digital System Design*, pages 271–275. IEEE, 2015.

[AJSvdP16] Waheed Ahmad, Marijn Jongerden, Mariëlle Stoelinga, and Jaco van de Pol. Model checking and evaluating QoS of batteries in MPSoC dataflow applications via hybrid automata. In *Proceedings of the 16th International Conference on Application of Concurrency to System Design*, pages 114–123. IEEE, 2016.

[ALP97] Marleen Adé, Rudy Lauwereins, and JA Peperstraete. Data memory minimisation for synchronous data flow graphs emulated on DSP-FPGA targets. In *Proceedings of the 34th Annual Design Automation Conference*, pages 64–69. ACM, 1997.

[AvdP16] Waheed Ahmad and Jaco van de Pol. Synthesizing energy-optimal controllers for multiprocessor dataflow applications with Uppaal Stratego. In *Proceedings of the International Symposium on Leveraging Applications of Formal Methods*, pages 94–113. Springer, 2016.

[BBDM00] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE*

Transactions on Very Large Scale Integration Systems, 8(3):299–316, 2000.

[BBHL95] Shuvra S Bhattacharyya, Joseph T Buck, Soonhoi Ha, and Edward A Lee. Generating compact code from dataflow specifications of multirate signal processing algorithms. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 42(3):138–150, 1995.

[BBLM10] Alessio Bonfietti, Luca Benini, Michele Lombardi, and Michela Milano. An efficient and complete approach for throughput-maximal SDF allocation and scheduling on multi-core platforms. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 897–902. European Design and Automation Association, 2010.

[BCM+92] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BDL+06] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Hakansson, Paul Petterson, Wang Yi, and Martijn Hendriks. UPPAAL 4.0. In *Proceedings of the Third International Conference on Quantitative Evaluation of Systems*, pages 125–126. IEEE, 2006.

[Bel58] RE Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[BELP96] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cycle-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, 1996.

[BFG+03] Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Konsta Karsisto. Survey of code-size reduction methods. *ACM Computing Surveys*, 35(3):223–267, 2003.

[Bjo04] D Bjorklund. Efficient code synthesis from synchronous dataflow graphs. In *Proceedings of the Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, pages 83–92. IEEE, 2004.

[BKKB02] Neal Bambha, Vida Kianzad, Mukul Khandelia, and Shuvra S Bhattacharyya. Intermediate representations for design automation of multiprocessor DSP systems. *Design Automation for Embedded Systems*, 7(4):307–323, 2002.

[BL93] Shuvra S Bhattacharyya and Edward A Lee. Scheduling synchronous dataflow graphs for efficient looping. *Journal of VLSI signal processing systems for signal, image and video technology*, 6(3):271–288, 1993.

[BL94] Shuvra S Bhattacharyya and Edward A Lee. Looped schedules for dataflow descriptions of multirate signal processing algorithms. *Formal Methods in System Design*, 5(3):183–205, 1994.

[BLMB09] Alessio Bonfietti, Michele Lombardi, Michela Milano, and Luca Benini. Throughput constraint for synchronous data flow graphs. In *Proceedings of the International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 26–40. Springer, 2009.

[BLMB13] Alessio Bonfietti, Michele Lombardi, Michela Milano, and Luca Benini. Maximum-throughput mapping of SDFGs on multi-core SoC platforms. *Journal of Parallel and Distributed Computing*, 73(10):1337–1350, 2013.

[BMKdD12] Bruno Bodin, Alix Munier-Kordon, and Benoît Dupont de Dinechin. K-periodic schedules for evaluating the maximum throughput of a synchronous dataflow graph. In *Proceedings of the International Conference on Embedded Computer Systems*, pages 152–159. IEEE, 2012.

[BML12] Shuvra S Bhattacharyya, Praveen K Murthy, and Edward A Lee. *Software synthesis from dataflow graphs*, volume 360. Springer Science & Business Media, 2012.

[BSL+02] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan, and Peter Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, pages 73–78. IEEE, 2002.

[CC10] Weijia Che and Karam S Chatha. Scheduling of synchronous data flow models on scratchpad memory based embedded processors. In *Proceedings of the International Conference on Computer-Aided Design*, pages 205–212. IEEE, 2010.

[CC13] Weijia Che and Karam S Chatha. Scheduling of synchronous data flow models onto scratchpad memory-based embedded processors. *ACM Transactions on Embedded Computing Systems*, 13(1s):30, 2013.

[Chr91] Philippe Chretienne. The basic cyclic scheduling problem with deadlines. *Discrete Applied Mathematics*, 30(2):109–123, 1991.

[CJSZ08] Louis-Claude Canon, Emmanuel Jeannot, Rizos Sakellariou, and Wei Zheng. Comparative evaluation of the robustness of dag scheduling heuristics. In *Grid Computing*, pages 73–84. Springer, 2008.

[COKH12] Junchul Choi, Hyunok Oh, Sungchan Kim, and Soonhoi Ha. Executing synchronous dataflow graphs on a SPM-based multicore architecture. In *Proceedings of the 49th Annual Design Automation Conference*, pages 664–671. ACM, 2012.

[Cor18] IBM Corporation. *CPLEX optimizer*. 2018. `https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer`.

[CS92] L-F Chao and EH-M Sha. Unfolding and retiming data-flow DSP programs for risc multiprocessor scheduling. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages 565–568. IEEE, 1992.

[CS97] Liang-Fang Chao and Edwin Hsing-Mean Sha. Scheduling data-flow graphs via retiming and unfolding. *IEEE Transactions on Parallel and Distributed Systems*, 8(12):1259–1267, 1997.

[CT14] Liang Chen and Mehdi Tahoori. Reliability-aware register binding for control-flow intensive designs. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6. IEEE, 2014.

[CZ12] Yuankai Chen and Hai Zhou. Buffer minimization in pipelined sdf scheduling on multi-core platforms. In *Proceedings of the 17th Asia

and South Pacific Design Automation Conference, pages 127–132. IEEE, 2012.

[DCdM+99] Koen Danckaert, Francky Catthoor, Hugo J de Man, Costas Goutis, and Konstantinos Masselos. Strategy for power-efficient design of parallel systems. *IEEE Transactions on Very Large Scale Integration Systems*, 7(2):258–265, 1999.

[DDL15] Xuan Khanh Do, Amira Dkhil, and Stéphane Louise. Self-timed periodic scheduling of data-dependent tasks in embedded streaming applications. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing*, pages 458–478. Springer, 2015.

[DDLR15] Amira Dkhil, Xuan Khanh Do, Stéphane Louise, and Christine Rochange. A hybrid scheduling algorithm based on self-timed and periodic scheduling for embedded streaming applications. In *Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 711–715. IEEE, 2015.

[DGGH92] SM Heemstra De Groot, Sabih H Gerez, and Otto E Herrmann. Range-chart-guided iterative data-flow graph scheduling. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 39(5):351–364, 1992.

[Dij76] EW Dijkstra. A discipline of programming. *Prentice-Hall Series in Automatic Computation*, 1976.

[DKV12] Anup Das, Akash Kumar, and Bharadwaj Veeravalli. Energy-aware communication and remapping of tasks for reliable multimedia multiprocessor systems. In *Proceedings of the 18th International Conference on Parallel and Distributed Systems*, pages 564–571. IEEE, 2012.

[DKV14] Anup Das, Akash Kumar, and Bharadwaj Veeravalli. Energy-aware task mapping and scheduling for reliable embedded computing systems. *ACM Transactions on Embedded Computing Systems*, 13(2s):72, 2014.

[DKV16] Anup Das, Akash Kumar, and Bharadwaj Veeravalli. Reliability and energy-aware mapping and scheduling of multimedia applications on

multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):869–884, 2016.

[DPNA13] Karol Desnos, Maxime Pelcat, Jean-François Nezan, and Slaheddine Aridhi. Pre-and post-scheduling memory allocation strategies on MPSoCs. In *Proceedings of the Electronic System Level Synthesis Conference*, pages 1–6. IEEE, 2013.

[DSB+12] Morteza Damavandpeyma, Sander Stuijk, Twan Basten, Marc Geilen, and Henk Corporaal. Modeling static-order schedules in synchronous dataflow graphs. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 775–780. EDA Consortium, 2012.

[DSB+13] Morteza Damavandpeyma, Sander Stuijk, Twan Basten, Marc Geilen, and Henk Corporaal. Schedule-extended synchronous dataflow graphs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(10):1495–1508, 2013.

[FHHG12] Mohammad H Foroozannejad, Trevor Hodges, Matin Hashemi, and Soheil Ghiasi. Postscheduling buffer management trade-offs in streaming software synthesis. *ACM Transactions on Design Automation of Electronic Systems*, 17(3):27, 2012.

[FKBS11] Sardar M Farhad, Yousun Ko, Bernd Burgstaller, and Bernhard Scholz. Orchestration by approximation: Mapping stream programs onto multicore architectures. *ACM SIGPLAN Notices*, 46(3):357–368, 2011.

[GBS05] Marc Geilen, Twan Basten, and Sander Stuijk. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Proceedings of the 42nd Annual Design Automation Conference*, pages 819–824. ACM, 2005.

[GGS+06] Amir Hossein Ghamarian, MCW Geilen, Sander Stuijk, Twan Basten, AJM Moonen, Marco JG Bekooij, Bart D Theelen, and MohammadReza Mousavi. Throughput analysis of synchronous data flow graphs. In *Proceedings of the Sixth International Conference on Application of Concurrency to System Design*, pages 25–36. IEEE, 2006.

[GTA06] Michael I Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *ACM SIGARCH Computer Architecture News*, 34(5):151–162, 2006.

[GZZ15] Yu-Lei Gu, Xue-Yang Zhu, and Guangquan Zhang. Pareto optimal scheduling of synchronous data flow graphs via parallel methods. In *Proceedings of the International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, pages 217–223. Springer, 2015.

[HFG13] Matin Hashemi, Mohammad H Foroozannejad, and Soheil Ghiasi. Throughput-memory footprint trade-off in synthesis of streaming software on embedded multiprocessors. *ACM Transactions on Embedded Computing Systems*, 13(3):46, 2013.

[HLO08] F Heras, J Larrosa, and A Oliveras. MiniMaxSAT: An efficient weighted Max-SAT solver. *Journal of Artificial Intelligence Research*, 31:1–32, 2008.

[HM93] Claire Hanen and Alix Munier. *Cyclic scheduling on parallel processors: An overview*. Rapports de recherche. Université de Paris-Sud, Centre d'Orsay, Laboratoire de Recherche en Informatique, 1993.

[Hol04] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*. Addison-Wesley Professional, 2004.

[HQR15] Maen Hammond, Guangzhi Qu, and Osamah A Rawashdeh. Deploying and scheduling vision based advanced driver assistance systems on heterogeneous multicore embedded platform. In *Proceedings of the Ninth International Conference on Frontier of Computer Science and Technology*, pages 172–177. IEEE, 2015.

[KBB06] Mukul Khandelia, Neal K Bambha, and Shuvra S Bhattacharyya. Contention-conscious transaction ordering in multiprocessor DSP systems. *IEEE Transactions on Signal Processing*, 54(2):556–569, 2006.

[KFH+08] Akash Kumar, Shakith Fernando, Yajun Ha, Bart Mesman, and Henk Corporaal. Multiprocessor systems synthesis for multiple use-cases of multiple applications on FPGA. *ACM Transactions on Design Automation of Electronic Systems*, 13(3):40, 2008.

[KMB04] Ming-Yung Ko, Praveen K Murthy, and Shuvra S Bhattacharyya. Compact procedural implementation in DSP software synthesis through recursive graph decomposition. *Proceedings of the 8th International Workshop on Software and Compilers for Embedded Systems*, pages 47–61, 2004.

[KMB07] Ming-Yung Ko, Praveen K Murthy, and Shuvra S Bhattacharyya. Beyond single-appearance schedules: Efficient DSP software synthesis using nested procedure calls. *ACM Transactions on Embedded Computing Systems*, 6(2):14, 2007.

[KMT+06] Akash Kumar, Bart Mesman, Bart Theelen, Henk Corporaal, and Ha Yajun. Resource manager for non-preemptive heterogeneous multiprocessor system-on-chip. In *Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, pages 33–38. IEEE, 2006.

[KTJR05] Rakesh Kumar, Dean M Tullsen, Norman P Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32–38, 2005.

[LF13] Marco Lattuada and Fabrizio Ferrandi. Modeling pipelined application with synchronous data flow graphs. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 49–55. IEEE, 2013.

[LGE12] Jing Lin, Andreas Gerstlauer, and Brian L Evans. Communication-aware heterogeneous multiprocessor mapping for real-time streaming systems. *Journal of Signal Processing Systems*, 69(3):279–291, 2012.

[LGY15] Weichen Liu, Zonghua Gu, and Yaoyao Ye. Efficient sat-based application mapping and scheduling on multiprocessor systems for throughput maximization. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 127–136. IEEE, 2015.

[LHCA88] Chung-Yee Lee, Jing-Jang Hwang, Yuan-Chieh Chow, and Frank D Anger. Multiprocessor scheduling with interprocessor communication delays. *Operations Research Letters*, 7(3):141–147, 1988.

[LKOH13] Chanhee Lee, Sungchan Kim, Hyunok Oh, and Soonhoi Ha. Failure-aware task scheduling of synchronous data flow graphs under real-time constraints. *Journal of Signal Processing Systems*, 73(2):201–212, 2013.

[LM+87a] Edward A Lee, David G Messerschmitt, et al. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[LM87b] Edward Ashford Lee and David G Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on computers*, 100(1):24–35, 1987.

[LO13] Daeyoung Lee and Hyunok Oh. A lifetime aware buffer assignment method for streaming applications on DRAM/PRAM hybrid memory. *ACM Transactions on Embedded Computing Systems*, 12(1s):36, 2013.

[LP93] Lori E Lucke and Keshab K Parhi. Data-flow transformations for critical path time reduction in high-level DSP synthesis. *IEEE transactions on computer-aided design of integrated circuits and systems*, 12(7):1063–1068, 1993.

[LRS83] Charles E Leiserson, Flavio M Rose, and James B Saxe. Optimizing synchronous circuitry by retiming. In *Proceedings of the Third Caltech Conference on Very Large Scale Integration*, pages 87–116. Springer, 1983.

[LS91] Charles E Leiserson and James B Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1-6):5–35, 1991.

[LSGE11] Jing Lin, Akshaya Srivatsa, Andreas Gerstlauer, and Brian L Evans. Heterogeneous multiprocessor mapping for real-time streaming systems. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, pages 1605–1608. IEEE, 2011.

[MAG12] Davit Mirzoyan, Benny Akesson, and Kees Goossens. Process-variation aware mapping of real-time streaming applications to MPSoCs for improved yield. In *Proceedings of the 13th International Symposium on Quality Electronic Design*, pages 41–48. IEEE, 2012.

[MB01] Praveen K Murthy and Shuvra S Bhattacharyya. Shared buffer implementations of signal processing systems using lifetime analysis

techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(2):177–198, 2001.

[MB04] Praveen K Murthy and Shuvra S Bhattacharyya. Buffer merging—a powerful technique for reducing memory requirements of synchronous dataflow specifications. *ACM Transactions on Design Automation of Electronic Systems*, 9(2):212–237, 2004.

[MBGS10] Orlando Moreira, Twan Basten, Marc Geilen, and Sander Stuijk. Buffer sizing for rate-optimal single-rate data-flow scheduling revisited. *IEEE Transactions on Computers*, 59(2):188–201, 2010.

[MBL97] Praveen K Murthy, Shuvra S Bhattacharyya, and Edward A Lee. Joint minimization of code and data for synchronous dataflow programs. *Formal Methods in System Design*, 11(1):41–70, 1997.

[MG13] Avinash Malik and David Gregg. Orchestrating stream graphs using model checking. *ACM Transactions on Architecture and Code Optimization*, 10(3):19, 2013.

[MG15] Avinash Malik and David Gregg. Heuristics on reachability trees for bicriteria scheduling of stream graphs on heterogeneous multiprocessor architectures. *ACM Transactions on Embedded Computing Systems*, 14(2):23, 2015.

[MS16] Mingze Ma and Rizos Sakellariou. Buffer minimization for rate-optimal scheduling of synchronous dataflow graphs on multicore systems. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing*, pages 325–340. Springer, 2016.

[MS17] Mingze Ma and Rizos Sakellariou. Code-size-aware mapping for synchronous dataflow graphs on multicore systems: Work-in-progress. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems Companion*, page 13. ACM, 2017.

[MS18a] Mingze Ma and Rizos Sakellariou. Communication-aware scheduling algorithms for synchronous dataflow graphs on multicore systems. In *Proceedings of the International Conference on Embedded Computer Systems*. ACM, 2018.

[MS18b] Mingze Ma and Rizos Sakellariou. Reducing code size in scheduling synchronous dataflow graphs on multicore systems. In *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, pages 57–62. ACM, 2018.

[NG93] Qi Ning and Guang R Gao. A novel framework of register allocation for software pipelining. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 29–42. ACM, 1993.

[ODH06] Hyunok Oh, Nikil Dutt, and Soonhoi Ha. Memory optimal single appearance schedule with dynamic loop count for synchronous dataflow graphs. In *Design Automation, 2006. Asia and South Pacific Conference on*, pages 6–pp. IEEE, 2006.

[PBL+95] JosC Luis Pino, Shuvra S Bhattacharyya, Edward Lee, et al. A hierarchical multiprocessor scheduling system for DSP applications. In *Conference Record of the Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 122–126. IEEE, 1995.

[PM91] Keshab K Parhi and David G Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *IEEE Transactions on Computers*, 40(2):178–195, 1991.

[Ram74] Chander Ramchandani. *Analysis of asynchronous concurrent systems by timed Petri nets*. PhD thesis, 1974.

[RC01] Kyoungseok Rha and Kiyoung Choi. Area-efficient buffer binding based on a novel two-port FIFO structure. In *Proceedings of the Ninth International Symposium on Hardware/Software Codesign*, pages 122–127. ACM, 2001.

[RS14] Kathrin Rosvall and Ingo 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.

[RS17] Kathrin Rosvall and Ingo Sander. Flexible and tradeoff-aware constraint-based design space exploration for streaming applications on heterogeneous platforms. *ACM Transactions on Design Automation of Electronic Systems*, 23(2):21, 2017.

[Rud93] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 42–47. IEEE, 1993.

[Rup18] Karl Rupp. 42 years of microprocessor trend data, 2018. https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/.

[SB09] Sundararajan Sriram and Shuvra S Bhattacharyya. *Embedded multiprocessors: Scheduling and synchronization*. CRC Press, 2009.

[SBGC07] Sander Stuijk, Twan Basten, MCW Geilen, and Henk Corporaal. Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proceedings of the 44th Annual Design Automation Conference*, pages 777–782. ACM, 2007.

[Sch97] Robert R Schaller. Moore's law: Past, present and future. *IEEE Spectrum*, 34(6):52–59, 1997.

[Sch06] Linus E Schrage. *Optimization modeling with LINGO*. Lindo System, 2006.

[SGB06a] Sander Stuijk, Marc Geilen, and Twan Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Proceedings of the 43rd Annual Design Automation Conference*, pages 899–904. ACM, 2006.

[SGB06b] Sander Stuijk, Marc Geilen, and Twan Basten. Sdf^3: SDF for free. In *Proceedings of the Sixth International Conference on Application of Concurrency to System Design*, pages 276–278. IEEE, 2006.

[Sha81] Micha Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.

[Sin07] Oliver Sinnen. *Task scheduling for parallel systems*, volume 60. John Wiley & Sons, 2007.

[SJ14] Mahendra Pratap Singh and Manoj Kumar Jain. Evolution of processor architecture in mobile phones. *International Journal of Computer Applications*, 90(4), 2014.

[SKH98] Wonyong Sung, Junedong Kim, and Soonhoi Ha. Memory efficient software synthesis from dataflow graph. In *Proceedings of the 11th International Symposium on System Synthesis*, pages 137–142. IEEE, 1998.

[SL93] Gilbert C Sih and Edward A Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, 1993.

[SMB+02] Greg Semeraro, Grigorios Magklis, Rajeev Balasubramonian, David H Albonesi, Sandhya Dwarkadas, and Michael L Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, pages 29–40. IEEE, 2002.

[SMG14] Radu Andrei Stefan, Anca Mariana Molnos, and Kees G. W. Goossens. dAElite: A TDM NoC supporting QoS, multicast, and fast connection set-up. *IEEE Transactions on Computers*, 63(3):583–594, 2014.

[SOH11] Tae-ho Shin, Hyunok Oh, and Soonhoi Ha. Minimizing buffer requirements for throughput constrained parallel execution of synchronous dataflow graph. In *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, pages 165–170. IEEE, 2011.

[Str18] StreamIt, 2018. `http://groups.csail.mit.edu/cag/streamit`.

[Stu07] Sander Stuijk. *Predictable mapping of streaming applications on multiprocessors*. PhD thesis, 2007.

[Sys18] Lindo Systems. *Lingo 17.0*. 2018. `https://www.lindo.com/index.php/products/lingo-and-optimization-modeling`.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[TBG+17a] Qi Tang, Twan Basten, Marc Geilen, Sander Stuijk, and Ji-Bo Wei. Mapping of synchronous dataflow graphs on mpsocs based on parallelism enhancement. *Journal of Parallel and Distributed Computing*, 101:79–91, 2017.

[TBG+17b] Qi Tang, Twan Basten, Marc Geilen, Sander Stuijk, and Ji-Bo Wei. Task-FIFO co-scheduling of streaming applications on MPSoCs with predictable memory hierarchy. *ACM Transactions on Embedded Computing Systems*, 16(2):49, 2017.

[Tea06] Gecode Team. Gecode: Generic constraint development environment, 2006. `https://www.gecode.org/`.

[Tei12] Jürgen Teich. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430, 2012.

[TGB+06] Bart D Theelen, Marc CW Geilen, Twan Basten, Jeroen PM Voeten, Stefan Valentin Gheorghita, and Sander Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Proceedings of the Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, pages 185–194. IEEE, 2006.

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

[TKA02] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the International Conference on Compiler Construction*, 2002.

[TZB98] Jürgen Teich, Eckart Zitzler, and Shuvra Bhattacharyya. Buffer memory optimization in DSP applications: An evolutionary approach. In *Proceedings of the International Conference on Parallel Problem Solving from Nature*, pages 885–894. Springer, 1998.

[WBJS06] Maarten Wiggers, Marco Bekooij, Pierre Jansen, and Gerard Smit. Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure. In *Proceedings of the Forth International Conference on Hardware/Software Codesign and System Synthesis*, pages 10–15. ACM, 2006.

[WH92] Duen-Jeng Wang and Yu Hen Hu. Fully static multiprocessor realization for real-time recursive DSP algorithms. In *Proceedings of the International Conference on Application Specific Array Processors*, pages 664–678. IEEE, 1992.

[WH94a] Duen-Jeng Wang and Yu Hen Hu. Fully static multiprocessor array realizability criteria for real-time recurrent DSP applications. *IEEE Transactions on Signal Processing*, 42(5):1288–1292, 1994.

[WH94b] Duen-Jeng Wang and Yu Hen Hu. Rate optimal scheduling of recursive dsp algorithms by unfolding. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 41(10):672–675, 1994.

[WK03] Frank Wolz and Reiner Kolla. Disproving the perfect-rate property of data-flow graphs unfolded by the least common multiple of the number of loop register. *IEEE Transactions on Computers*, 52(5):688, 2003.

[WLW+10] Yi Wang, Duo Liu, Meng Wang, Zhiwei Qin, and Zili Shao. Optimal task scheduling by removing inter-core communication overhead for streaming applications on MPSoC. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 195–204. IEEE, 2010.

[WM95] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[Yan10] Xin-She Yang. *Nature-inspired metaheuristic algorithms*. Luniver Press, 2010.

[YF97] Tao Yang and Cong Fu. Heuristic algorithms for scheduling iterative task computations on distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 8(6):608–622, 1997.

[ZDP+13] Zheng Zhou, Karol Desnos, Maxime Pelcat, Jean-François Nezan, William Plishker, and Shuvra S Bhattacharyya. Scheduling of parallelized synchronous dataflow actors. In *Proceedings of the International Symposium on System on Chip*, pages 1–10. IEEE, 2013.

[ZGBS12] Xue-Yang Zhu, Marc Geilen, Twan Basten, and Sander Stuijk. Static rate-optimal scheduling of multirate DSP algorithms via retiming and unfolding. In *Proceedings of the 18th Real-Time and Embedded Technology and Applications Symposium*, pages 109–118. IEEE, 2012.

[ZGBS14] Xue-Yang Zhu, Marc Geilen, Twan Basten, and Sander Stuijk. Memory-constrained static rate-optimal scheduling of synchronous dataflow graphs via retiming. In *Proceedings of the Conference on Design, Automation & Test in Europe*, page 325. European Design and Automation Association, 2014.

[ZGBS16] Xue-Yang Zhu, Marc Geilen, Twan Basten, and Sander Stuijk. Multiconstraint static scheduling of synchronous dataflow graphs via retiming and unfolding. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(6):905–918, 2016.

[ZPB+11] George F Zaki, William Plishker, Shuvra S Bhattacharyya, Charles Clancy, and John Kuykendall. Vectorization and mapping of software defined radio applications on heterogeneous multi-processor platforms. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, pages 31–36. IEEE, 2011.

[ZPB+13] George F Zaki, William Plishker, Shuvra S Bhattacharyya, Charles Clancy, and John Kuykendall. Integration of dataflow-based heterogeneous multiprocessor scheduling techniques in GNU radio. *Journal of Signal Processing Systems*, 70(2):177–191, 2013.

[ZPB+16] Zheng Zhou, William Plishker, Shuvra S Bhattacharyya, Karol Desnos, Maxime Pelcat, and Jean-Francois Nezan. Scheduling of parallelized synchronous dataflow actors for multicore signal processing. *Journal of Signal Processing Systems*, 83(3):309–328, 2016.

[ZSJ09] Jun Zhu, Ingo Sander, and Axel Jantsch. Buffer minimization of real-time streaming applications scheduling on hybrid CPU/FPGA architectures. In

Proceedings of the Conference on Design, Automation and Test in Europe, pages 1506–1511. European Design and Automation Association, 2009.

[ZSJ10] Jun Zhu, Ingo Sander, and Axel Jantsch. Constrained global scheduling of streaming applications on MPSoCs. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 223–228. IEEE, 2010.

[ZTB00] Eckart Zitzler, Jürgen Teich, and SS Bhattclcharyya. Evolutionary algorithms for the synthesis of embedded software. *IEEE Transactions on Very Large Scale Integration Systems*, 8(4):452–455, 2000.

[ZYG+15] Xue-Yang Zhu, Rongjie Yan, Yu-Lei Gu, Jian Zhang, Wenhui Zhang, and Guangquan Zhang. Static optimal scheduling for synchronous data flow graphs with model checking. In *Proceedings of the International Symposium on Formal Methods*, pages 551–569. Springer, 2015.