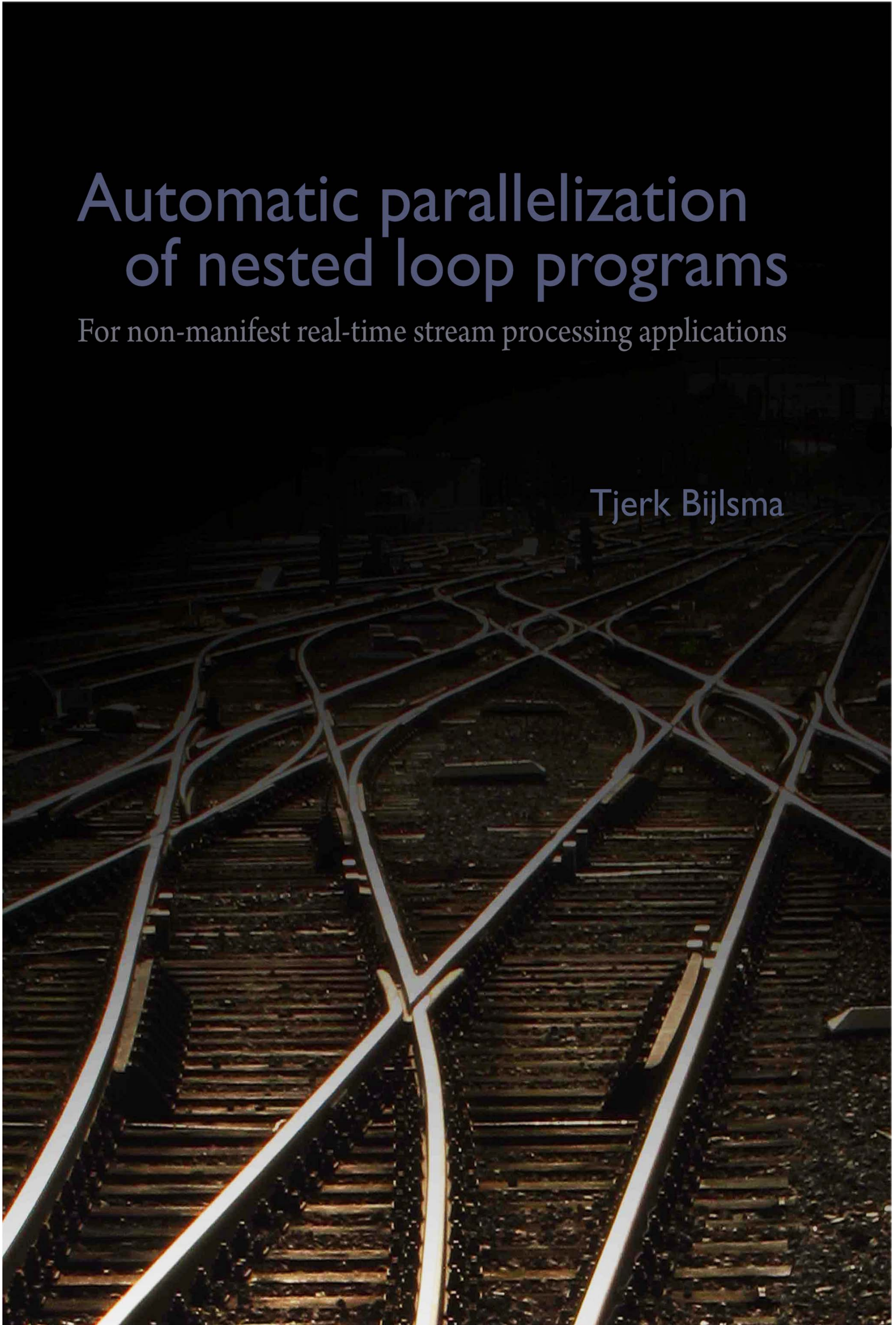


Automatic parallelization of nested loop programs

For non-manifest real-time stream processing applications

Tjerk Bijlsma

AUTOMATIC PARALLELIZATION of NESTED LOOP PROGRAMS for NON-MANIFEST REAL-TIME STREAM PROCESSING APPLICATIONS

Tjerk Bijlsma

Members of the dissertation committee:

Prof.dr.ir.	M.J.G. Bekooij	University of Twente (first promotor)
		NXP Semiconductors
Prof.dr.ir.	G.J.M. Smit	University of Twente (second promotor)
Prof.dr.ir.	R. Leupers	RWTH Aachen University
Prof.dr.ir.	H. Corporaal	Eindhoven University of Technology
Dr.	T.P. Stefanov	Leiden University
Prof.dr.ir.	A. Rensink	University of Twente
Prof.dr.ir.	J.L. Hurink	University of Twente
Prof.dr.ir.	A.J. Mouthaan	University of Twente (chairman)

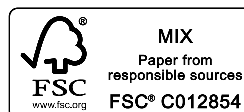
This work was carried out at NXP Semiconductors in a project of NXP Semiconductors Research. This work contributed to the Center for Telematics and Information Technology (CTIT) research program.

Copyright © 2011 By T. Bijlsma, Oss, The Netherlands.

All rights reserved. No part of this book may be reproduced or transmitted, in any form or by any means, electronic or mechanical, including photocopying, microfilming, and recording, or by any information storage or retrieval system, without the prior written permission by the author.

Cover design by M. Bijlsma. This thesis was printed by Gildeprint, The Netherlands.

ISBN 978-90-365-3173-3
ISSN 1381-3617, No. 11-189
DOI 10.3990/1.9789036531733



AUTOMATIC PARALLELIZATION of NESTED LOOP PROGRAMS for NON-MANIFEST REAL-TIME STREAM PROCESSING APPLICATIONS

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 1 juli 2011 om 14.45 uur

door

Tjerk Bijlsma

geboren op 23 oktober 1981
te Franeker

Dit proefschrift is goedgekeurd door de promotoren:

prof. dr. ir. M. J. G. Bekooij
en
prof. dr. ir. G. J. M. Smit

Voorwoord

Dit proefschrift is ontstaan met de hulp, steun en ook afleiding van veel collega's, vrienden en familie. Hiervoor wil ik hen bedanken.

Ik begin met het bedanken van Pierre Jansen, Gerard Smit en Pascal Wolkotte. Tijdens mijn afstuderen bij Pierre heb ik samen met Gerard en Pascal een artikel geschreven. Het onderzoek voor mijn afstudeeropdracht en het schrijven van dit artikel bevielen zo goed dat ik op zoek gegaan ben naar een promotieplek. Via Gerard en Pierre vond ik een mogelijkheid bij Philips Research, wat vrij snel NXP Semiconductors Research werd. Gerard is mijn tweede promotor geworden en heeft mij de afgelopen vier jaar begeleid. Ik wil Gerard en Pierre bedanken voor alle discussies en hun commentaar.

Bij NXP kwam ik in het Hijdra project, waar Marco Bekooij mijn dagelijkse begeleider en later mijn eerste promotor werd. Samen hebben we gewerkt aan het onderzoek voor en de ontwikkeling van de multiprocessor compiler en de parallellisatie tool Omphale. De resultaten hebben we op meerdere conferenties en workshops gepresenteerd. Ik wil Marco bedanken voor alle discussies, inzichten en zijn commentaar waarbij we vaak dankbaar gebruik maakten van de whiteboards die naast de koffieautomaten hingen.

Very important for a promotion are the office mates. Aleksandar Milutinovic and Benny Åkesson, thanks for sharing five different offices with me in the past four years. Furthermore, I want to thank all colleagues and fellow Ph.D. students in the SOC Architectures and Infrastructure group at NXP, especially Maarten Wiggers and Erik Larsson, for the interesting discussions and refreshing coffee brakes. Our multiprocessor compiler would not have been what it currently is, without the contributions of the students Sven Goossens, Stefan Geuns, and Joost Hausmans.

Ik wil ook de collega's in de Computer Architecture for Embedded Systems groep van de Universiteit Twente bedanken. Bedankt voor alle gezellige discussies en voor de bureaus die jullie met me wilden delen terwijl ik Enschede bezocht. De secretaresses Marlous, Thelma en Nicole wil ik bedanken voor alle hulp.

Hiel belangryk binne de freonen en de famylje, sûnder jim' wie ik nea sa fier kommen. Bedankt foar alle belangstelling en ôflieding troch de jierren hinne. Us heit en mem wol ik bedanke, om't se my fan jongs ôf oan steund en motivearre ha mei alles wat ik die. Myn twa broerkes Folkert en Menno wol ik bedanke foar alle ôflieding en gekkichheden, mar ek om't se my altiten holpen ha wêr't se mar koene. Menno, bedankt foar it moaie omslach. Lieme en Aly, bedankt foar al jimme belangstelling en om't ik by jimme altiten wolkom bin, as of't in twadde thús is. Mar boppe al wol ik Dieuwke bedanke foar al har leafde en geduld de ôfrûne jierren.

Abstract

This thesis is concerned with the automatic parallelization of real-time stream processing applications, such that they can be executed on embedded multiprocessor systems.

Stream processing applications can be encountered in the channel decoding and video decoding domain. These applications typically have real-time requirements. Important trends for stream processing applications are that they become more computational intensive and that they contain more non-manifest conditions and expressions. For non-manifest conditions and expressions, the behavior is unknown at compile time.

Stream processing applications are often executed on multiprocessor systems. Because stream processing applications become more computational intensive, the number of processors in these systems increases. Due to the increasing number of processors, the mapping effort for stream processing applications onto these systems increases. Furthermore, the validation effort for stream processing applications on such systems increases, because the satisfaction of temporal constraints has to be validated for an application that is executed on multiple processors.

To map stream processing applications onto a multiprocessor system, we use a multiprocessor compiler. We consider multiprocessor compilers that perform automatic parallelization. Otherwise, the user should perform the partitioning of the application manually, which can be time-consuming and error-prone.

For our application domain, we focus on the extraction of function parallelism, because it is often available in stream processing applications. Stream processing applications contain function parallelism, because they are often composed of functions that can be executed independently from each other.

The extraction of parallelism requires the derivation of data dependencies in an application. These data dependencies indicate the execution order of the tasks. If the data dependencies between two statements cannot be determined at compile time, then these statements can often not be executed in parallel.

The parallel execution of tasks requires inter-task communication buffers. Many approaches use first-in-first-out (FIFO) buffers for the inter-task communication. A FIFO buffer supports only one reading and one writing task. However, an application from which parallelism is extracted may contain multiple statements that read from and write into an array. To replace the communication via such an array by inter-task communication, multiple FIFO buffers should be used. It can be difficult to extract a function for a task that determines per array access from which FIFO buffer the value should be read or into which buffer it should be written.

To verify that the temporal constraint of a stream processing application is met, a temporal analysis model should be derived. Such a model should be extracted from an application that contains cyclic data dependencies and non-manifest behavior.

Current parallelization approaches have difficulties with the extraction of function parallelism from stream processing applications. Some of these approaches require applications with manifest behavior and affine index-expressions. For these applications, they can derive data dependencies and insert inter-task communication via FIFO buffers. But, these approaches cannot support stream processing applications with non-manifest loops. Furthermore, current approaches can only extract a temporal analysis model from applications with manifest behavior and without cyclic data dependencies.

To address the issues mentioned above, we present in this thesis an automatic parallelization approach to extract function parallelism from sequential descriptions of real-time stream processing applications. We introduce a language to describe stream processing applications. The key property of this language is that all dependencies can be derived at compile time. In our language we support non-manifest loops, if-statements, and index-expressions. We introduce a new buffer type that can always be used to replace the array communication. This buffer supports multiple reading and writing tasks. Because we can always derive the data dependencies and always replace the array communication by communication via a buffer, we can always extract the available function parallelism. Furthermore, our parallelization approach uses an underlying temporal analysis model, in which we capture the inter-task synchronization. With this analysis model, we can compute system settings and perform optimizations. Our parallelization approach is implemented in a multiprocessor compiler. We evaluated our approach, by extracting parallelism from a WLAN channel decoder application and a JPEG decoder application with our multiprocessor compiler.

Samenvatting

Dit proefschrift gaat over automatische parallellisatie van realtime stroomverwerkende applicaties, opdat ze uitgevoerd kunnen worden op een embedded systeem met meerdere rekenkernen.

Stroomverwerkende applicaties komen vaak voor in het kanaaldecoderings- en videodecoderingsdomein en verwerken een continue stroom van waardes. Dit soort applicaties hebben meestal realtime eisen. Belangrijke trends voor stroomverwerkende applicaties zijn dat ze berekeningsintensiever worden en dat ze steeds meer non-manifeste condities en expressies bevatten. Bij non-manifest condities en expressies is het gedrag onbekend gedurende de compilatie.

Stroomverwerkende applicaties worden vaak uitgevoerd op systemen met meerdere rekenkernen. Omdat stroomverwerkende applicaties rekenintensiever worden, neemt het aantal rekenkernen in deze systemen toe. Door het toenemende aantal rekenkernen wordt het steeds moeilijker om stroomverwerkende applicaties op deze systemen af te beelden. Ook wordt het moeilijker om het gedrag van de stroomverwerkende applicaties te valideren voor systemen met meerdere rekenkernen, omdat voor een applicatie die door meerdere rekenkernen uitgevoerd wordt, gevalideerd moet worden dat aan de tijd gerelateerd eisen voldaan wordt.

Om stroomverwerkende applicaties af te beelden op een systeem met meerdere rekenkernen gebruiken we een multiprocessorcompiler. We beschouwen multiprocessorcompilers die automatische parallellisatie uitvoeren. Anders zou de gebruiker het opdelen van de applicatie handmatig uit moeten voeren, wat tijd rovend en foutgevoelig is.

Voor ons applicatiedomein richten we ons op het afleiden van functieparallellisme, omdat dit vaak beschikbaar is in stroomverwerkende applicaties. Stroomverwerkende applicaties bevatten functieparallellisme, omdat ze vaak zijn opgebouwd uit functies die onafhankelijk van elkaar uitgevoerd kunnen worden.

Het afleiden van parallellisme vereist dat de data-afhankelijkheden in een applicatie bepaald worden. Deze data-afhankelijkheden geven de volgorde aan waarin de taken

uitgevoerd moeten worden. Als de data-afhankelijkheden tussen twee statements niet bepaald kunnen worden gedurende de compilatie, dan kunnen deze statements niet parallel uitgevoerd worden.

Het parallel uitvoeren van taken vereist buffers voor de communicatie tussen de taken. Veel aanpakken gebruiken first-in-first-out (FIFO) buffers, waarvoor geldt dat de eerst geschreven waarde als eerste gelezen wordt. Een FIFO buffer ondersteunt slechts één enkele lezende en schrijvende taak. Een applicatie waarvan parallellisme afgeleid wordt, kan meerdere statements bevatten die schrijven in of lezen uit een array. Het vervangen van de communicatie via deze array door communicatie tussen taken, vereist het gebruik van meerdere FIFO buffers. Het kan moeilijk zijn om een functie af te leiden, die per schrijf– of leesactie voor een array bepaalt in welke FIFO buffer de waarde geschreven of gelezen zou moeten worden.

Om te verifiëren dat aan de tijd gerelateerde eisen van een stroomverwerkende applicatie voldaan wordt, moeten we een temporeel analysemodel afleiden. Dit model moet afgeleid kunnen worden van een applicatie die cyclische data-afhankelijkheden en non-manifest gedrag bevat.

De huidige parallellisatie-aanpakken hebben moeite met het afleiden van functieparallellisme van stroomverwerkende applicaties. Sommige van deze aanpakken vereisen applicaties met manifest gedrag en affine indexexpressies. Voor deze applicaties kunnen ze de data-afhankelijkheden afleiden en communicatie tussen de taken invoegen via FIFO buffers. Deze aanpakken kunnen stroomverwerkende applicaties met non-manifeste lussen niet ondersteunen. Verder kunnen de huidige aanpakken alleen een temporeel analysemodel afleiden van applicaties met manifest gedrag en zonder cyclische data-afhankelijkheden.

De boven genoemde problemen beschouwende, presenteren wij in dit proefschrift een automatische parallellisatie-aanpak om functieparallellisme af te leiden van sequentiële beschrijvingen van realtime stroomverwerkende applicaties. We introduceren een taal om stroomverwerkende applicaties te beschrijven. De kern eigenschap van deze taal is dat alle data-afhankelijkheden bepaald kunnen worden gedurende de compilatie. In onze taal ondersteunen we non-manifeste lussen, if-statements, en indexexpressies. We introduceren een nieuw buffertype dat altijd toegepast kan worden om de communicatie via een array te vervangen. Dit buffertype ondersteunt meerdere schrijvende en lezende taken. Omdat we altijd de data-afhankelijkheden af kunnen leiden en altijd de array communicatie kunnen vervangen door communicatie via een buffer, kunnen we altijd het beschikbare functieparallellisme afleiden. Onze parallellisatie-aanpak gebruikt een onderliggend temporeel analysemodel, waarin we de synchronisatie tussen de taken weergeven. Met dit analysemodel kunnen we instellingen voor een systeem met meerdere rekenkernen berekenen en optimalisaties uitvoeren. Onze parallellisatie-aanpak is geïmplementeerd in een multiprocessorcompiler. We hebben onze aanpak geëvalueerd door met onze multiprocessorcompiler parallelisme af te leiden van een WLAN-kanaaldecodeerapplicatie en een JPEG-decodeerapplicatie.

Contents

1 Introduction **1**

1.1 Stream processing application domain 2
1.2 Embedded multiprocessor system 3
1.3 Multiprocessor compiler 4
1.4 Problem statement 8
1.5 Contributions 9
1.6 Justification of the approach 9
1.7 Outline 10

2 Related work **11**

2.1 Sequential programming languages 12
2.2 Parallelization tools 14
2.3 Parallel programming languages 17
2.4 Temporal analysis models 19
2.5 Conclusion 20

3 Multiprocessor compiler **23**

3.1 Multiprocessor compiler overview 23
3.2 Overview of the parallelization phase 26
3.3 Conclusions 30

4 Dependency graph extraction **31**

4.1 Data dependencies 32
4.2 Data dependency analysis 33
4.3 Omphale input language 37
4.4 Dependency graph 44

4.5 Access pattern extraction . . . 48
4.6 Conclusion . . . 50

5 Inter-task communication buffers 51
5.1 Memory consistency model . . . 52
5.2 FIFO communication issues . . . 55
5.3 Buffer with sliding windows . . . 59
5.4 Buffer with overlapping windows . . . 64
5.5 Conclusion . . . 69

6 Communication and synchronization insertion 71
6.1 Generic template . . . 72
6.2 Manifest access patterns . . . 74
6.3 Non-manifest access patterns . . . 85
6.4 Stream processing . . . 91
6.5 Access types . . . 94
6.6 Conclusion . . . 97

7 Temporal analysis model 99
7.1 Shortcomings of local analysis . . . 100
7.2 The CSDF analysis model . . . 104
7.3 Synchronization sections . . . 106
7.4 Modeling manifest synchronization behavior . . . 114
7.5 Modeling non-manifest synchronization behavior . . . 123
7.6 Conclusion . . . 124

8 Case studies 125
8.1 WLAN channel decoder . . . 126
8.2 JPEG decoder . . . 132
8.3 Conclusion . . . 137

9 Conclusion 139
9.1 Summary . . . 140
9.2 Contributions . . . 142
9.3 Future work . . . 142

Bibliography 145

List of publications 155

CHAPTER 1

Introduction

This thesis is concerned with the automatic parallelization of real-time stream processing applications, such that they can be executed on embedded multiprocessor systems. These stream processing applications process streams of values, contain if-statements and while-loops with conditions that depend upon input values from the stream, and often have temporal requirements. State-of-the-art automatic parallelization approaches can extract parallelism from such applications, but have difficulties supporting if-statements and while-loops. Furthermore, these approaches have difficulties with the extraction of a temporal analysis model, which is required for the verification of temporal constraints.

In this thesis, we will present a new parallelization approach for stream processing applications. We will introduce a new language that supports if-statements and while-loops, such that we can always analyze the dependencies. We introduce a new buffer type that we can always use to replace the communication via arrays, such that parallelism can always be extracted. Besides a task graph, we extract an analysis model from the synchronization behavior between the tasks in the task graph. With this analysis model, the temporal requirements can be verified and even optimizations can be computed for the task graph. This parallelization approach is part of a multiprocessor compiler with which we demonstrated the extraction of parallelism given industrial relevant stream processing applications.

The organization of this chapter is as follows. In section 1.1, we will first examine the characteristics of stream processing applications, followed by a discussion on the targeted embedded multiprocessor systems in Section 1.2. For the multiprocessor compilers that compiles stream processing applications for multiprocessor systems, we will discuss the input format and the state-of-the-art automatic parallelization approaches, in Section 1.3. Considering the requirements and the state-of-the-art for automatic paral-

lelization, Section 1.4 presents our problem statement. Subsequently, we present our key contributions in section 1.5. Section 1.6 gives a justification of our approach. An outline of the remainder of the thesis is presented in Section 1.7.

1.1 Stream processing application domain

In this thesis, we consider so called stream processing applications that process endless streams of input values. Stream processing applications are encountered in the channel decoding and the video processing domain and are therefore considered interesting by NXP semiconductors, for which this research was partly conducted.

A stream processing application processes an endless stream of input values. Therefore, these applications typically contain an *endless loop* in which these values are read and processed.

The behavior of a stream processing application can depend upon its input values, because it may contain an if-statement or a while-loop with a condition for which the result of its expression depends upon input values from the stream. In addition, the result of an index-expression used to access array elements may also depend upon input values. The result of such a condition or index-expression cannot be evaluated at compile time and they are therefore called *non-manifest*. In contrast, for a manifest condition or index-expression the result can be evaluated at compile time. We will call if-statements and while-loops with conditions that cannot be evaluated at compile time *non-manifest if-statements* and *non-manifest loops*. Index-expressions that cannot be evaluated at compile time will be called *non-manifest index-expressions*. We will call the combination of non-manifest loops, if-statements, and index-expressions *non-manifest statements*.

For stream processing applications, we identify two trends: 1) they become more computational intensive and 2) they tend to contain an increasing number of non-manifest conditions and expressions. The computational requirements for stream processing applications increase, due to the increased quantity and quality of the media that the users desire. For channel decoding and video processing applications, this often implies an increase in bit rate, which requires more computations to be performed for a single value from the input stream. Stream processing applications tend to contain more non-manifest conditions and expressions, such that they can adapt to changes in their environment.

A *channel decoder* performs stream processing and contains non-manifest conditions and has therefore been an important driver application behind this work. Figure 1.1 depicts a channel decoder. This channel decoder processes an *endless stream of input values* in the form of samples that it receives in buffer s_b from the radio front-end task t_f. The channel decoder either decodes by executing the tasks t_{dec}, t_{p0}, and t_{p1} or detects by executing t_{det}. Depending on the state of the previous iteration stored in buffer s_c, the tasks t_{dec}, t_{p0}, t_{p1}, and t_{det} determine if they have to execute.

Often, a stream processing application has a real-time constraint in the form of a *temporal requirement*. Such a requirement typically comes in the form of a throughput constraint. Channel decoders typically have a throughput constraint, such that it is guaranteed that the input values are consumed at a fixed rate. This prevents the buffers at

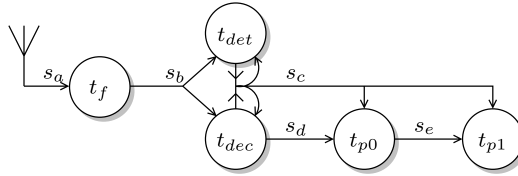


Figure 1.1: A task graph of a channel decoder that performs stream processing

the input of the channel decoder from overflowing, such that no value will be lost. Note that a throughput requirement does not restrict the end-to-end latency of an application. Therefore, the application can be pipelined to meet the throughput requirement.

1.2 Embedded multiprocessor system

To execute a stream processing application, we will use an embedded multiprocessor system. These systems typically contain multiple processors and memories. We will program an embedded multiprocessor system with a so called shared address space programming model that allows tasks to communicate by writing in and reading from shared memory locations.

For embedded systems, we identify two trends: 1) the numbers of processors on a chip is increasing, and 2) multiple memories are used on a single chip. The first trend indicates that instead of increasing the frequency of a processor, multiple processors are provided, such that the total amount of processing power is increased. Furthermore, due to the increasing number of processors, providing a single physical shared memory on the chip would become a performance bottleneck. Therefore, the trend is to provide multiple memories on a chip, which results in the processors having a non-uniform memory access (NUMA) latency. Often, some of these memories are assigned as a local memory to some processors, which typically implies that the memory cannot be read by all processors on the chip.

To program an embedded multiprocessors system, we will consider a *shared address space* programming model [CGS99] in which tasks can communicate and synchronize via shared locations in the memories. This programming model comes with some additional communication overhead, because to write at a memory location, the address of the location has to be communicated along with the value to be written. The clear benefit of this programming model is that it simplifies the mapping of tasks to processors, because the memory accesses of tasks are independent from the processor the task is mapped to. Each processor will use the same address for a shared location, such that it is not necessary to tailor the code of a task for its processor.

Figure 1.2 depicts two multiprocessor systems with a *physical shared address space*, on top of which the shared address space programming model can easily be used. In these architectures, the tasks can perform communication and synchronization via the shared memory. Note, that physical shared address space architectures can contain local

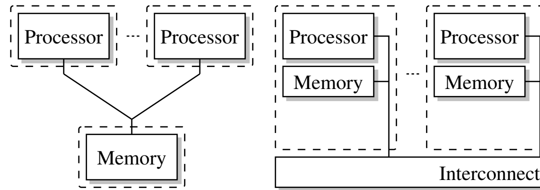


Figure 1.2: Examples of embedded multiprocessor systems with a physical shared address space

memories that cannot be read by all processors. If a task needs to read from locations in such a local memory, it should be executed by a processor that can also read these locations.

1.3 Multiprocessor compiler

A multiprocessor compiler maps a stream processing application onto an embedded multiprocessor system. Depending upon its input, the compiler has to perform automatic parallelization. We will discuss the consequence of starting from a parallel or a sequential description of an application. Subsequently, we examine state-of-the-art automatic parallelization approaches for stream processing applications.

1.3.1 Multiprocessor compiler input

A multiprocessor compiler can either start from a parallel or sequential description of an application. By starting from a sequential description, a user can provide a compact description of the application, without the need to partition the application manually. In addition, it is beneficial that the parallelization front-end of the compiler automatically extracts an analysis model, such that temporal requirements can be validated.

For the execution of stream processing applications on embedded multiprocessor systems, the trends of the applications and the systems have two clear implications: the mapping effort of applications onto the multiprocessor system increases and the validation effort of the mapped applications increases [MP03]. The *mapping effort* of a stream processing application onto a multiprocessor system increases, because the required processing power for the application can only be satisfied by using multiple processors. Therefore, the application has to be partitioned into multiple tasks that can be executed in parallel on different processors. Because these tasks are executed on different processors, they should perform inter-task communication via shared memory. The *validation effort* increases, because the satisfaction of the temporal requirements has to be validated for a stream processing application that is executed over multiple processors that share resources.

To execute a stream processing application on an embedded multiprocessor system, a multiprocessor compiler is required. This compiler should compute a mapping of tasks

to processors and allocate memories to tasks, such that the temporal requirements of the application are satisfied.

A multiprocessor compiler can start from a parallel description or a sequential description of the stream processing application. A sequential description of an application specifies the execution order of the statements, such that it does not require synchronization statements. A parallel description does not completely specify the execution order, such that it must contain synchronization statements. By starting from a *parallel description*, the user has to manually partition the application and can also optimize this description. Starting from a parallel description simplifies the front-end of the compiler, because it does not have to extract parallelism. A drawback of this approach is that the user has to partition the application manually, which can be time-consuming and error-prone. The granularity of the tasks in such a manual partitioning of an application is often optimized for a specific multiprocessor system. Furthermore, such a description typically requires manual annotations of the identified tasks, which requires the user to have expert knowledge of the system.

We will start from a *sequential description* of an application from which the compiler automatically extracts parallelism. This requires automatic parallelization. We define automatic parallelization as the identification of tasks and the insertion of inter-task communication and synchronization statements. Therefore, we derive the dependencies in a sequential description, such that tasks can be identified and the array communication for which these dependencies have been derived can be replaced by inter-task communication and synchronization statements. Due to the automatic parallelization, a compact sequential description of the application can be given as input by a user. Such a compact sequential description is typically easy to debug and does not require the user to have expert knowledge of the system. Therefore, we think it is desirable that the front-end of a multiprocessor compiler performs automatic parallelization.

To verify that the temporal constraints of a stream processing application are satisfied, either a corresponding analysis model is required or the constraints should be verified by iterative validation. Verification by iteratively executing the application is time consuming and can in general not guarantee that the temporal constraint will always be satisfied, due to the resource sharing between the processors. An analysis model can be used to guarantee the temporal constraints to be satisfied and can also be used to compute the required resources.

For a compiler, a temporal analysis model of an application can be provided as an input, or can be automatically extracted. Providing an analysis model as *input*, requires the user to describe the synchronization behavior of the parallelized stream processing application, which requires expert knowledge of both the used model and the partitioned application. Furthermore, manually describing this synchronization behavior is error-prone and time consuming. It is beneficial that such an analysis model is *automatically extracted*, such that it corresponds to the parallel description of the application.

1.3.2 Automatic parallelization

In this thesis, we will focus on automatic parallelization for stream processing applications that have temporal requirements.

To exploit the parallel execution of the processors in a multiprocessor system, two types of parallelism can be identified: function and data parallelism [CGS99, ALSU06]. For *data parallelism*, typically independent loop-iterations are identified, where each iteration is executed on a different processor, resulting in load balancing of the computations and a reduced synchronization overhead. However, the drawback is that there are often dependencies between loop iterations and after executing the loop iterations, the results need to be combined, which may require additional control-statements and synchronization.

We chose to focus on *function parallelism*, for which a task is created from each function call in the code. Function parallelism can often be extracted from stream processing applications, if they are composed of functions that can be executed independently from each other, as shown by our driver application in Figure 1.1. The main reason to extract function parallelism is that the execution of a stream processing application can be pipelined, such that potentially its throughput can be increased.

A task graph is extracted from the sequential description of a stream processing application. The sequential description of such an application is often in the form of a nested loop program (NLP), where the bodies of the loops contain assignment-statements with function calls. In an NLP, typically the assignment-statements read from and write into arrays. A task graph is extracted by creating a task from each assignment-statement. These tasks share dependencies via the arrays that they write and read.

To execute the tasks from the task graph in parallel, they have to be extended to use the programming model of the target multiprocessor system. The array communication of the NLP is replaced by inter-task communication via buffers. Synchronization statements are inserted into the tasks, such that the dependencies between the tasks are maintained. This ensures that values are written before they are read.

We have identified two state-of-the-art parallelization approaches for stream processing applications. The Compaan/PN approach [MNS10, Ste04, TKD02, TKD04a, TKD04b, VNS07] extracts function parallelism from NLPs. Therefore, each assignment-statement is executed in a separate task. Sprint [CDVS07] is the second state-of-the-art approach, which extracts parallelism from annotated C code.

The Compaan/PN approach extracts parallelism from affine NLPs and inserts first-in-first-out (FIFO) buffers for the inter-task communication. An *affine* NLP only contains index-expressions that are a summation of variables multiplied by constants plus an optional constant, e.g. $3i + j - 4$. The expressions for the bounds of the for-loops may contain parameters. The value of such a parameter can be unknown at compile time, but it will not change during the execution of the for-loop. Due to the affine index-expressions, exact data dependencies can be extracted [Fea91], such that the NLP can be transformed into single assignment (SA) form. For an NLP in SA form, a location in an array is written at most once during an execution of the NLP. Inter-task communication via FIFO buffers can be inserted into such an NLP.

In [Ste04] the Compaan/PN approach is extended to extract parallelism from weakly dynamic NLPs. A weakly dynamic NLP may contain non-manifest if-statements in combination with manifest for-loops. A task that is extracted from an assignment-statement in the body of a non-manifest if-statement, has to perform inter-task communication via special FIFO buffers that contain controllers for reading and writing. These controllers are aware of the exact execution schedule of the tasks. The Compaan/PN approach does not support the extraction of parallelism from non-manifest loops in stream processing applications with arbitrary index-expressions.

While this thesis was written, the Compaan/PN approach has been extended to support weakly dynamic NLPs with non-manifest for-loops [NNS10]. However, this extension does still not support the extraction of parallelism from non-manifest while-loops in stream processing applications with non-affine index-expressions.

The Sprint approach extracts parallelism from annotated C code, but requires user input for non-manifest loops or if-statements. This approach can insert FIFO buffers for dependencies between tasks that it can analyze. If the dependencies cannot be analyzed, due to for example non-manifest loops or if-statements, the user should manually insert synchronization and communication statements. Manual insertion of synchronization statements can be error-prone and can lead to hard to debug race-conditions [AB09].

Besides the extraction of a task graph, an analysis model is required, such that temporal requirements can be verified. For the Compaan/PN approach, methods exist to extract an analysis model [HT07, MNS10], but these are only applicable for acyclic task graphs for which they can verify the performance, instead of computing the required resources to meet the temporal constraint. In addition, both approaches cannot capture the non-manifest behavior of our driver application in an analysis model.

Table 1.1 depicts the capabilities of the discussed state-of-the-art approaches and the requirements that we identified for the extraction of parallelism from stream processing applications. To extract parallelism from stream processing applications, the *Compaan/PN* approach supports affine index-expressions and non-manifest if-statements and for-loops with non-manifest upper and lower bounds. This approach can only extract an analysis model from a manifest acyclic task graph. The *Sprint* approach requires user input, if non-manifest expressions are encountered that cannot be analyzed and does not extract an analysis model. For the parallelization of stream processing applications, the support of non-manifest index-expressions is desirable and for non-manifest loops and if-statements a must, as illustrated by our driver application in Figure 1.1. Support of parameterization can be desirable, because this can result in a more efficient task graph. The

Table 1.1: Automatic parallelization approaches compared to the identified requirements

	Non-manifest index-expressions	Non-manifest control-statements	Parameterization	Temporal analysis model
Compaan/PN	Affine	If-statements and for-loops	Yes	Manifest acyclic NLPs
Sprint	User input	User input	No	None
Required for stream processing	Desirable	Loops and if-statements	Desirable	Must

extraction of an analysis model is a must, because it is required to validate the temporal requirements of the application.

1.4 Problem statement

The problem addressed in this thesis is to extract a task graph from the NLP of a stream processing application that may contain non-manifest if-statements, loops, and index-expressions and the extraction of a temporal analysis model from this task graph, such that we can guarantee that the temporal constraints of the stream processing application are satisfied and that this task graph can be executed on an embedded multiprocessor system.

The first part of the problem, is the extraction of parallelism from the NLP of an application. Before a task can be created from each assignment-statement, the dependencies between these tasks should be identified. These dependencies indicate the execution order of the tasks, such that values shared between the tasks will be written before they will be read. If the dependencies between the tasks cannot be analyzed, the execution of the tasks cannot be pipelined. Furthermore, if there are dependencies that may point to any location in memory, the local memories in a multiprocessor system cannot be used, because these cannot be read from each processor.

To execute the tasks from a task graph on a multiprocessor system, the communication via arrays in an NLP should be replaced by communication via buffers. A lot of approaches replace array communication by FIFO buffers [CCS+08, CDVS07, DHRA06, RFGEL08, TKD04b, VNS07], but this may introduce the so called buffer selection problem and reordering problem. If multiple tasks read from and write into an array and this array has to be replaced by multiple FIFO buffers, the *buffer selection problem* occurs. Conditions have to be extracted and added to the tasks. These conditions determine if values have to be written in or read from the FIFO buffers. The buffer selection problem will be discussed in detail in Section 5.2.3. If a FIFO buffer is applied for an array that has different access patterns for reading and writing, the *reordering problem* occurs. Values are read from a FIFO buffer in the order that they are written into it. If the writing task writes its values sequentially into the FIFO buffer, the reading task requires a reordering task and reorder memory, to read the locations in the order of its access pattern. The reordering problem will be discussed in detail in Section 5.2.2.

The third problem is the extraction of a temporal analysis model from the synchronization behavior of the task graph, such that the temporal requirements can be verified. The extracted model should be temporally conservative, which means that the modeled events do not occur earlier as they occur in the task graph, in order to give guarantees for the temporal behavior. Furthermore, these events might occur in non-manifest if-statements or non-manifest loops, which should be modeled conservatively.

1.5 Contributions

Our main contribution is the introduction of a new automatic parallelization approach, for stream processing applications that are described by NLPs. We defined a new language that supports non-manifest statements and from which we can always extract the dependencies. For this language, we introduced a kind of SA form that we use as a requirement for the description of our applications. We can always extract parallelism, because we introduced a new buffer type that can always be used to replace the array communication from which the dependencies have been derived. We can extract a temporal analysis model from the synchronization between the tasks in the task graph, by conservatively modeling this synchronization.

In more detail, our contributions are:

1. The introduction of a new language that supports non-manifest statements to describe stream processing applications, such that we can always extract data dependencies and function parallelism (Chapter 4).

2. We can always extract parallelism and guarantee deadlock-free execution, because we introduce two new buffer types that support multiple reading and writing tasks and non-manifest access patterns (Chapter 5). The first buffer type can always be applied, because it can handle latency critical cyclic data dependencies, and the other has a low synchronization overhead.

3. We can extract a temporal analysis model from the inter-task synchronization of stream processing applications, except for some non-manifest latency critical cyclic data dependencies. The used buffer types and the inserted inter-task synchronization enable the extraction of a temporal analysis model. With the extracted analysis model, we can guarantee satisfaction of the temporal requirements and perform optimizations (Chapter 7).

4. Our parallelization techniques have been integrated in a multiprocessor compiler for stream processing applications, with which we extracted parallelism from industrial relevant applications (Chapter 3 and 8).

1.6 Justification of the approach

The automatic parallelization approach presented in this thesis, has been constructed using a bottom-up approach. We used this approach, because it allowed us to present solutions during the development of the multiprocessor compiler.

Due to the bottom-up approach, we defined our own Omphale input language (OIL) for the compiler. This language has been generalized step by step. At each point during development, we could analyze all dependencies in an application and therefore extract parallelism from OIL, because we had strict control of the semantics and statements supported by our language. Backward compatibility with the semantics of existing languages was considered less important.

During the development of the compiler, the addition of new statements and semantics to OIL has triggered generalizations for the buffer types. Furthermore, modeling techniques were defined, such that a conservative analysis model could be extracted and resource requirements could be computed.

1.7 Outline

The organization of this thesis is as follows. We will start with discussing related automatic parallelization approaches. With this discussion, we will highlight our contributions in relation to existing work. The next chapter will present an overview of our multiprocessor compiler for stream processing applications, which relates to contribution 4. This compiler accepts a stream processing application described sequentially in OIL together with a temporal requirement and returns an executable for our multiprocessor system. After the overview, we will discuss our automatic parallelization approach in detail. First, we discuss the dependency analysis, the syntax, and the semantics of OIL, which corresponds to contribution 1. From an application described in OIL, we can always extract a dependency graph. To replace the array communication from which these dependencies were derived by buffers, two new buffer types are introduced, as claimed in contribution 2. We will present templates that define the placement of synchronization and communication statements for these buffers into the tasks, such that the dependency graph is transformed into a task graph. From the synchronization behavior between the tasks in this task graph a temporal analysis model can be extracted, corresponding to contribution 3. We can use this analysis model to validate temporal requirements for the task graph, or even to compute optimizations. With the presented automatic parallelization approach, two case studies have been performed to illustrate its possibilities, this partly relates to contribution 4. Amongst others, the parallelization of a WLAN channel decoder is demonstrated for which we can compute buffer capacities that maximize the overlap in task executions.

The outline of this thesis is as follows. Chapter 2 discusses the related automatic parallelization approaches. Subsequently, an overview of our multiprocessor compiler is given in Chapter 3. In Chapter 4, we discusses the dependency analysis and the extraction of a dependency graph from OIL. To replace the communication via arrays in the dependency graph, Chapter 5 introduces two buffer types. The insertion of statements into the tasks to use these buffers is presented in Chapter 6, which results in a task graph. The extraction of a conservative analysis model from such a task graph is the last step for our automatic parallelization approach and is presented in Chapter 7. Chapter 8 presents an industrial relevant case study that we use to evaluate our parallelization tool. In Chapter 9, we will present our conclusions.

CHAPTER 2

Related work

Abstract - This chapter examines related parallelization approaches, such that we can highlight the differences with our approach. We examine sequential programming languages that can be used as input, parallelization tools, parallel programming languages that can be used as output, and temporal analysis models.

In this chapter, we will highlight the differences between our automatic parallelization approach and related approaches. We will consider four issues: the used sequential programming language to describe input applications, the used parallelization tool, the used temporal analysis model, and the used parallel programming language to describe the task graphs that can be extracted by parallelization tools.

What a parallelization tool carries out is depicted in Figure 2.1. A parallelization tool accepts an application described in a sequential programming language as input and returns a task graph described in a parallel programming language. We want to explicitly differentiate between sequential and parallel programming languages. Therefore, we define a *sequential programming language* as a language that does not include inter-task synchronization or communication statements. In contrast, we define a *parallel programming language* as a language that contains explicit inter-task synchronization or communication statements.

The outcome of a comparison with related approaches is that we see three key differentiators, between our approach and related approaches. The first differentiator is that we can always analyze the data dependencies in OIL, also for non-manifest state-

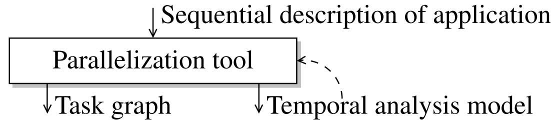


Figure 2.1: Overview of a parallelization approach

ments, such that only the data dependencies will determine the execution order of the tasks. Our second differentiator is that we introduce a new buffer type that can always be inserted to replace the communication via an array. The third differentiator is the underlying temporal analysis model for our parallelization approach, which is used to reduce the synchronization overhead and to compute sufficient buffer sizes, such that temporal constraints are satisfied.

Though this chapter will highlight the differences between our automatic parallelization approach and related approaches, we recognize the value of the related approaches.

For this chapter, the outline is as follows. First, Section 2.1 discusses the sequential programming languages. In section 2.2, we discuss related parallelization tools. We will discuss the parallel programming languages that can be used for a task graph that is extracted by a parallelization approach, in Section 2.3. The underlying temporal analysis models that are suitable for a parallelization approach are discussed, in Section 2.4. Finally, Section 2.5 presents the conclusions.

2.1 Sequential programming languages

The sequential programming language that is used to describe the input application of an automatic parallelization tool determines the parallelism that this tool can extract. Essential for the extraction of parallelism is the dependency analysis. If data dependencies can be derived, then they can be used to specify the execution order between the assignment-statements that should at least be maintained. Our requirements for a sequential programming language are that it supports non-manifest loops and if-statements, that the data dependencies are explicit, and that the data dependencies can be derived. This should result in a language in which it is transparent for the programmer which assignment-statements will be executed in parallel. If the data dependencies can be derived, the execution order of the tasks in the task graph will only be determined by these dependencies.

For the discussion of related sequential programming languages, we will try to classify them. Therefore, we will identify pure declarative languages, pure applicative languages, and pure imperative languages. We define a *pure declarative language* to only describe the logic of a computation, but not how the result should be computed. A *pure applicative language* is defined as a language in which the data dependencies between the assignment-statements are encoded, but not the execution order of the assignment-statements. In such a language, the data dependencies will determine the possible execution orders of the assignment-statements. We define a *pure imperative language* as a

language in which an explicit execution order of the assignment-statements is specified and the data dependencies are implicit.

DSWP [ORS+06], Cordes [CMM10], FP-MAP [KC97], and MAPS [CCS+08] extract parallelism from applications described in C. The C language contains a lot of elements from an imperative language, because C requires control-statements that describe an execution order for the assignment-statements in the application. Due to this explicit execution order, these languages can support non-manifest statements. But, the C language also contains elements from an applicative language, because it is possible to express data dependencies that can be derived by these parallelization approaches. However, since the semantics of the C language do not enforce the data dependencies to be explicit, the dependency analysis is not always successful. If the data dependencies cannot be derived, certain statements may not be executed in parallel, to prevent incorrect functional behavior.

In addition to the approaches in the previous paragraph, Sprint [CDVS07] extracts parallelism from applications described in the C language that are annotated with pragmas to indicate parallelization points. Adding pragmas to the C language results in a language that contains elements from an applicative language. These pragmas indicate points at which data dependencies can be derived that might not have been automatically identified by the parallelization approach.

The language Compel [TE68] is an applicative language. The execution order of the statements is only determined by the data dependencies, which are explicit, such that they can always be extracted. The data dependencies can completely determine the execution order, because Compel is a SA language. For an application written in a SA language, during the lifetime of a variable, the variable is assigned a value only once. Therefore, no additional dependencies are required to prevent values from being overwritten. However, because this language does not contain control-statements, it is difficult to describe the behavior of stream processing applications that have non-manifest statements.

The Sil [KvMN+92] and Silage [HRG+90, VSR96] languages contain a lot of elements from an applicative language. In these languages, an application is described by a directed graph that describes the dependencies between the functions. The power of these languages is that they are SA languages, such that the parallel execution of the statements is completely determined by the data dependencies. In addition to Compel, these languages do support a form of control-statements, such that they contain some imperative elements. Therefore, it may be possible to express stream processing applications using these languages.

The Compaan/PN [Tur07, VNS07, Ste04, NNS10] and Phideo [LvMvdW+91] parallelization approaches require an application in the form of an affine NLP as input. An affine NLP is described in a language in which assignment-statements can be nested in for-loops with manifest bounds, where the extensions in [Ste04, NNS10] also support non-manifest if-statements and for-loops with non-manifest bounds. The assignment-statements in the NLPs may access arrays, but must use affine index-expressions. The control-flow determines the execution order of an assignment-statement, which is an element from an imperative language. An element from an applicative language is that all data dependencies can be derived between the accesses of array elements, due to the

affine index-expressions and the supported loops. Because all data dependencies can be analyzed, these approaches can transform the affine NLPs into SA form, which is an essential step for these approaches. In this transformed NLP, the execution order between assignment-statements will only be determined by the data dependencies. However, the non-manifest behavior of our stream processing applications cannot be expressed in this language.

In this thesis, we introduce a new language to describe stream processing applications. Our language OIL contains aspects from an imperative language, because an application may contain assignment-statements nested in non-manifest loops or if-statements. An application described in OIL contains assignment-statements nested in loops and if-statements, such that initially at least a valid execution order of the assignment-statements is specified that will certainly not result in deadlock for the application. An applicative aspect of OIL is that we can always extract data dependencies. For non-manifest statements, we can at least extract approximated data dependencies [BCF97] between accesses of arrays, whereas for manifest statements we can extract data dependencies between the accesses of array elements. Between assignment-statements, the execution order is only determined by the data dependencies, because we require an application to be described in our own SA form. Though some approaches consider the transformation into SA form as an essential step, we did not encounter problems while describing our stream processing applications in our own SA form. Because we can always analyze the data dependencies and only these dependencies determine the execution order between assignment-statements, we can always extract function parallelism.

2.2 Parallelization tools

Automatic parallelization is performed in the front-end of a multiprocessor compiler and extracts a task graph described in a parallel programming language from an application described in a sequential programming language. The first step is the identification of tasks in the sequential programming language and the extraction of dependencies between these tasks, which results in a dependency graph. Next, the dependencies are replaced by inserting inter-task communication and synchronization statements into the tasks, such that we get a task graph from which the tasks can be executed in parallel on a multiprocessor system. Our requirement for a parallelization tool is that it can automatically extract all function parallelism from an input application that may contain non-manifest loops and if-statements, where the execution order of the tasks will only be determined by the extracted data dependencies.

In this section, we will first examine parallelization tools that extract *data parallelism* and subsequently we will examine tools that extract *function parallelism*. With the discussions in this section, we want to emphasize the possibility of our parallelization tool to extract function parallelism from stream processing applications, as claimed in contribution 1.

Data parallelism The approaches presented in [Ban94, Bas03, CG06, Fea96, WL91] can be applied to extract data parallelism from affine NLPs. They require affine index-

expressions in an NLP, such that all data dependencies for the array accesses can be captured in the polyhedral model [Fea91, Pug94, PW94]. The goal of these approaches is to minimize the dependencies between loop iterations, such that each iteration can be assigned to a different task. Using the polyhedral model, they can compute transformations for the data dependencies in an NLP, amongst others to maintain only data dependencies between the iterations of the outer-most loop. If this is possible, there is only synchronization between the iterations of the outer-most loop, such that the iterations of nested loops can be executed in parallel.

Extensions to the approaches in the previous paragraph have been proposed, such that data parallelism can be extracted from while loops. In [LG95], depending on the dependencies in the loop nest, it may be possible to transform the loops in the NLP, such that the while loop becomes the outer most loop, for which the iterations can be executed in parallel. Collard [Col95] extends this approach, by allowing speculative executions of while loop iterations, where an unnecessary executed loop iteration can be undone. Both approaches require affine index-expressions, such that they can capture the dependencies in the polyhedral model.

In [RP95] an approach is presented to extract data parallelism from non-manifest loops, for which speculative execution of loop iterations is performed, without requiring affine expressions. But, this approach does not perform transformations to the loop nest, instead it only determines if the dependencies allow iterations to be executed in parallel.

The tools presented by Franke [FO05] and Meijer [MKTdK07] can be used to extract data parallelism from affine NLPs with manifest loops, such that the data dependencies can be captured in the polyhedral model, with which transformations for the loop nests can be computed. In addition, Franke inserts *barrier statements* [CGS99] into the tasks for the inter-task synchronization. One barrier is used for the synchronization of multiple tasks. If a task executes a barrier statement, it is blocked until the other tasks that synchronize on this barrier executed their barrier statement. Meijer inserts FIFO buffers for the data dependencies between iterations.

For stream processing applications, it is often not possible to extract iterations without dependencies, which may limit the amount of data parallelism that can be extracted. Furthermore, stream processing applications often contain non-manifest statements for which the dependencies cannot be captured in the polyhedral model, such that only a limited number of transformations can be applied to increase the available data parallelism. Nonetheless, we see the extraction of data parallelism from stream processing applications as interesting future work.

The description of a stream processing application from the channel decoding or video processing domain often indicates a functional partitioning [Wal91, IEE07, SP09] that already expresses the pipeline to process an endless stream of input values. Such a functional partitioning implicitly specifies the available function parallelism. The extracting of function parallelism fits well with these applications, because each function can be assigned to a separate processor to construct the pipeline to process the endless stream of input values. Therefore, the remainder of this section will focus on tools to extract function parallelism.

Function parallelism Sprint [CDVS07] extracts function parallelism from C-code. *Pragmas* are used to indicate the potential parallelization points. Sprint inserts FIFO buffers for the inter-task communication. They can identify the reordering and buffers selection problems, but require the user to manually solve them by inserting synchronization statements and shared memory communication.

Thies [TCA07] presents an approach to pipeline the execution of C-code. This approach requires the user to insert pragmas into the code and restricts the pipelining to the outermost loop in the code, where the dependencies must be acyclic. Between the different stages of the pipeline, all shared values are copied in FIFO order.

Sprint and the approach presented by Thies require the user to indicate the parallelization points, because they cannot always derive the data dependencies. We think that it is desirable that the parallelization tool can derive the data dependencies in the sequential programming language, such that the parallelism can be extracted automatically.

FP-MAP [KC97] and MAPS [CCS+08] extract function parallelism from C-code. Based upon a dependency analysis, assignment-statements are assigned to tasks. FP-MAP does not insert inter-task synchronization statements into the tasks. MAPS inserts FIFO buffers for the inter-task communication, but cannot solve the reordering and buffer selection problems, such that this may limit the amount of extracted function parallelism.

DSWP [ORS+06] and Cordes [CMM10] extract function parallelism from C-code inspired by software pipelining. Based on the control-flow and the data dependencies, basic blocks [ALSU06] are identified in the C-code. These blocks can be executed in parallel and for the dependencies between the basic blocks, barrier synchronization is inserted. But, the dependencies between the basic blocks are not only determined by the data dependencies, but also by sequence dependencies due to the control-flow of an application. Therefore, the extracted function parallelism may be limited.

Compel [TE68], Sil [KvMN+92], Silage [HRG+90, VSR96], and Sisal [FCO90, GSH88] are applicative languages. One of the main goals of these languages is the extraction of function parallelism. These languages are single assignment (SA) languages [TE68] from which all data dependencies can be derived, such that all function parallelism can be extracted. However, the compilers for these languages do not insert inter-task communication and synchronization statements. Instead, they compute a valid execution order for the tasks at compile time. At run-time, this execution order should be enforced by a central scheduling unit. In contrast, using synchronization statements inside the tasks does not enforce a single execution order and does not require a central scheduling unit, which can become a bottleneck in current multiprocessor systems.

The Compaan/PN parallelization tools [Tur07, VNS07] can extract parallelism from affine NLPs. Due to the affine index-expressions and the manifest loops, data dependencies can be extracted. Using the data dependencies, a task graph can be extracted in which the array accesses are replaced by inter-task communication via FIFO buffers. Solutions have been presented to solve the reordering and buffer selection problems [TKD04b, TKD02], but these are only applicable for affine index-expressions and manifest loops and if-statements.

CompaanDyn [SD03] is an extension of the Compaan/PN parallelization tools that supports non-manifest if-statements. A task extracted from an assignment-statement that

is nested in a non-manifest if-statement has to perform inter-task communication via a special FIFO buffer. For this FIFO buffer, a special controller is added to the producer and the consumer of the buffer. At compile time, these controllers are configured by computing a valid execution order of the tasks that communicate via the FIFO buffer. Therefore, the execution of the tasks is not only determined by the data dependencies, but also by sequence dependencies, due to non-manifest if-statements. Furthermore, CompaanDyn still does not support non-manifest loops.

While this thesis was written, the CompaanDyn approach has been extended to support weakly dynamic NLPs with non-manifest for-loops [NNS10]. These for-loops may contain a non-manifest upper or lower bound. However, this extension does still not support non-manifest while-loops and non-affine index-expressions. After publishing the draft version of this thesis we were told that the PN tool supports stream processing applications with an endless loop, however how these loops are supported has not been published. Because the CompaanDyn approach provides a solution for only a part of the problems addressed in this thesis, it is not possible to give a fair comparison of the results of this approach and the results of the approach presented in this thesis.

Our automatic parallelization tool Omphale can extract function parallelism from stream processing applications that may contain non-manifest statements. From applications described in OIL we can at least extract approximated data dependencies [BCF97]. Using these data dependencies, we can replace the array accesses by inter-task communication and synchronization via one of our new buffer types. Because we use the data dependencies to insert the inter-task synchronization, only the data dependencies determine the execution order of the tasks. Furthermore, if we use the new buffer type that is suitable for latency critical cycles, a value that has been written into this buffer is immediately available for reading, such that maximum function parallelism is extracted.

2.3 Parallel programming languages

In this section, we will discuss parallel programming languages. These languages contain inter-task communication and synchronization statements, such that they are suitable to describe a task graph that has to be executed on a multiprocessor system. The requirement that we have for a parallel programming language is that it contains communication and synchronization calls that are rich enough, such that we can circumvent the buffer selection and reordering problems.

Examples of parallel programming languages are Cilk++ [Lei09], MPI [Mes03], OpenMP [DM98, Ope08], Pthreads [Pth96], StreamIt [DHRA06], and the language proposed by Reid et al. [RFGEL08]. In these languages, the user explicitly indicates the parallelism in an application by inserting pragmas for the potential parallelization points. Furthermore, data dependencies should be encoded by inserting synchronization statements. In these languages, both data and function parallelism can be expressed. In Cilk++, MPI, and OpenMP the amount of extracted data parallelism can be determined at runtime, depending on the current workload of the system. A brief overview of these approaches can be found in [DM98, MHT+10].

To express function parallelism in Cilk++, MPI, OpenMP, Pthreads, StreamIt, or the language proposed by Reid et al., the user has to insert inter-task synchronization statements for the shared variables. In addition, if the execution of the tasks has to be pipelined, the user has to insert special inter-task communication statements to encode the pipelined execution of the tasks and compute the size of the used buffers manually. These languages do not provide synchronization and communication calls to use so called read and write windows of which you can have an arbitrary number in a buffer, such that the user has to solve the buffer selection problem.

The synchronization and communication statements that are called by the tasks in a task graph can be specified by an application programming interface (API) that is implemented by a streaming library, as e.g. TTL [vdWdKH+04], which includes the interfaces specified in [dKSvdW+00, NKG+02, RvEP02]. By using such an API in a sequential programming language, according to our definition, it becomes a parallel programming language. The API of TTL provides synchronization statements to the user, such that a buffer with a sliding read and write window can be used. Such a window contains a number of consecutive locations that can be accessed in an arbitrary order. But, the TTL API only supports that all locations in a window are either available for reading or writing. For a cyclic task graph this may cause deadlock, as we will show in Section 5.3.3. Furthermore, TTL does not support multiple producers, such that it does not avoid the buffer selection problem.

The approach for inter-task communication proposed in [HGT07] uses *containers* for the inter-task communication, where a container is a place holder for values. Inside a container, locations can be accessed in any order and therefore a reordering task is not required. After values are written in a container, the container is released such that the consumer can read from it. But, released containers have to be read in FIFO order, such that there is no reordering between released containers. Furthermore, this approach only allows complete containers to be released and therefore a container should not be released before the values in the container have been written.

Parallel programming using one of the discussed parallel programming languages requires the user to manually partition the application and to explicitly define the synchronization that has to be performed. This allows the user to apply some optimizations to the task graph. However, the drawback is that the user is often responsible for the insertion of synchronization for shared variables, where incorrect synchronization may lead to hard to debug race-conditions [AB09, Lee06]. Therefore, we chose to perform automatic parallelization, starting from an application described in a sequential programming language.

The presented parallel programming languages and APIs provide inter-task communication and synchronization statements that cannot be used in a straightforward way to avoid the buffer selection problem. Furthermore, these approaches do not provide synchronization statements to overlap multiple read and write windows, such that deadlock may be introduced for cyclic task graphs. Therefore, we introduce a new buffer type with associated inter-task communication and synchronization statements. This buffer type avoids the reordering and buffer selection problems and can be applied for latency

critical data dependencies. In Section 5.4, we will prove that this buffer type can always be inserted to replace the array communication in an application that is described in OIL.

2.4 Temporal analysis models

In this section, we will focus on the underlying temporal analysis model for our parallelization approach, which we claimed to have for our approach in contribution 3. We will discuss two possible underlying temporal analysis models that can be used to guarantee the satisfaction of a throughput constraint for a stream processing application. Next, available approaches to extract an analysis model from a parallel programming language are examined.

Analysis models The synchronous languages Signal, Lustre, and Esterel have an underlying analysis model [BCE+03]. But, the goal of the underlying analysis model is not the verification of a throughput constraint, instead they require the model to reason formally about the functional behavior of their stream processing applications.

The analysis models known to us that can be used to guarantee a throughput constraint are the real-time calculus [TCN00] and dataflow analysis [Wig09] models. The *real-time calculus* analysis model can be used to provide guarantees on the throughput constraint and buffer capacities for a stream processing application that will be executed on a multiprocessor system. We define a buffer capacity as the number of locations available in a buffer. Input to the real-time calculus approach is a characterization of the executions of the tasks in the task graph, which requires worst-case and best-case execution times for the tasks. However, this analysis technique is currently less suitable for cyclic task graphs, whereas we target applications that may have a cyclic task graph. An example of such a cyclic task graph is given in Figure 1.1.

We chose to use a *dataflow analysis model* as an underlying analysis model of our automatic parallelization approach, because it supports cycles in the analysis model. The cyclo static dataflow (CSDF) model is suitable to model periodic synchronization behavior between tasks via their buffers, even for cyclic task graphs. In addition, the CSDF model also enables our automatic parallelization tool to perform optimizations, for example by computing a buffer capacity that may be smaller than the communicated array, by selecting synchronization calls that can be combined [HBC11], or by selecting the buffer type to be inserted [Geu10]. Note that these optimizations use the required throughput as a constraint.

Analysis model extraction Several approaches address the extraction of an analysis model from a task graph. In [HT07, HKH+09], the extraction of a real-time calculus analysis model from a task graph is presented. They consider applications that only contain FIFO buffers for the inter-task communication. The buffer capacities and other parameters of the system are required before the real-time calculus model can be used to verify if the application will meet its throughput constraint. Furthermore, the used analysis model restricts the task graphs to be acyclic.

In addition to the Compaan/PN approach, Meijer proposes to model a task graph using a dataflow model [MNS10]. With the dataflow model, the throughput of a task graph

is analyzed and tasks that can be merged are selected. This approach is only capable to extract a dataflow model from an acyclic task graph in which tasks communicate via FIFO buffers.

The extraction of a CSDF model from tasks that perform inter-task communication via buffers with sliding windows is presented in [DBC+07]. This approach discusses the extraction of a CSDF model from a sliding read and write window in a buffer and also presents a modeling technique to model multiple sliding read windows and one sliding write window in a buffer. But, this paper only focuses at a single buffer and therefore does not discuss the extraction of a CSDF model from the inter-task communication in a task graph.

In contrast to the presented approaches, our approach does not only extract a temporal analysis model if the task graph is in a suitable form. Instead, we constructed OIL such that we can extract a task graph from it, from which we can capture the inter-task synchronization in a CSDF model. Because we start from a sequential programming language, we insert the synchronization statements in such a way into the tasks that their synchronization behavior can be captured in the CSDF model. For a manifest application, we can always capture the synchronization in a CSDF model. For non-manifest behavior, we perform unconditional synchronization that we can capture in a CSDF model, unless the application contains a too latency critical cyclic data dependency. For such a latency critical cyclic data dependency, the modeled synchronization behavior can be too conservative, such that the CSDF model deadlocks and is not suitable for temporal analysis. To the best of our knowledge, no related automatic parallelization approach inserts inter-task synchronization statements into the tasks, such that a temporal analysis model can be extracted, despite non-manifest statements.

For our approach, the underlying CSDF model enables the computation of system settings, such as buffer capacities and scheduler settings, that are required to satisfy a given throughput constraint. Furthermore, the underlying model can be used for other optimizations, e.g. to select synchronization calls that can be combined [HBC11] or to select the buffer type that should be inserted [Geu10].

2.5 Conclusion

In this chapter, we have discussed sequential programming languages, parallelization tools, parallel programming languages, and temporal analysis models. For these four topics, we have discussed the state-of-the-art approaches and how they relate to our approach.

In the sequential programming languages used as input by parallelization tools, the analysis of dependencies determines the parallelism that can be extracted. In the languages that contain assignment-statements nested in loops, all data dependencies can be derived, but non-manifest statements are not supported. Therefore, we introduced our language OIL. In OIL we can describe stream processing applications and we can always derive the data dependencies, such that we can always extract function parallelism.

Current parallelization approaches can extract data parallelism or function parallelism from stream processing applications. We focused on the extraction of function parallelism, because stream processing applications typically contain function parallelism. For the related parallelization approaches, we found that none of them can guarantee that a similar level of parallelism is automatically extracted from applications with non-manifest statements.

An existing parallel programming language to specify the inter-task communication and synchronization, could be considered for the output of a parallelization approach. From the discussed approaches, we concluded that none of them provides a sufficient rich API that allows us to circumvent the buffer selection and reordering problems in a straightforward way.

We examined the available temporal analysis models and concluded that the dataflow model is the most suitable for the considered stream processing applications, because it supports cyclic task graphs. None of the existing approaches can extract a dataflow model from a stream processing application with cyclic data dependencies. Our parallelization tool extracts a task graph from a sequential description in OIL, such that the inserted inter-task synchronization can be captured in a dataflow model. This dataflow model can be used to compute buffer capacities to meet a given throughput constraint.

CHAPTER 3

Multiprocessor compiler

Abstract - In this chapter, we present an overview of the internals of our multiprocessor compiler that we implemented for the research that is presented in this thesis. After presenting the phases of our multiprocessor compiler, we present a more detailed overview of the parallelization phase by presenting its subphases.

In this chapter, we will present an overview of the internals of our multiprocessor compiler for real-time stream processing applications. This compiler has been implemented for the research that we present in this thesis. Our compiler starts from a sequential description of a real-time application and returns an executable that can be executed on our multiprocessor system. We will discuss the four phases of our multiprocessor compiler. Subsequently, we will present a more detailed overview of the automatic parallelization phase.

The outline of this chapter is as follows. First, Section 3.1 presents an overview of our multiprocessor compiler. Subsequently, we will discuss the parallelization phase in more detail, in Section 3.2. We will present our conclusions, in Section 3.3.

3.1 Multiprocessor compiler overview

Our multiprocessor compiler creates an executable, from an application described in a sequential programming language. As input for our compiler, we consider a real-time application with a temporal requirement. Due to this temporal requirement, we may

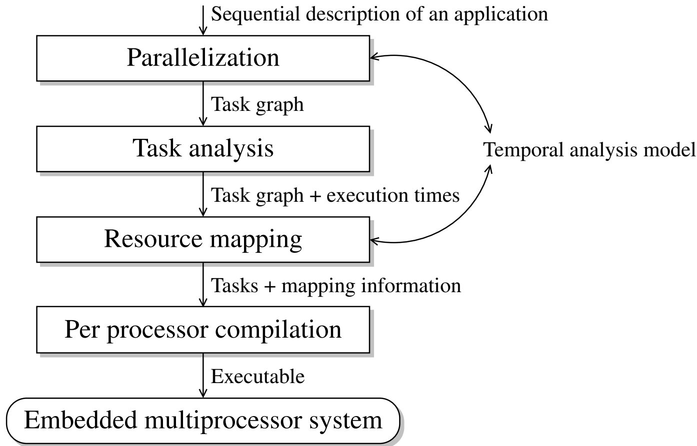


Figure 3.1: A multiprocessor compiler for real-time stream processing applications

have to use multiple processors for the execution of this application. In this section, we will discuss the four phases in our multiprocessor compiler that create an executable from such a real-time application. This executable can be executed on our multiprocessor system.

Figure 3.1 depicts the phases in our multiprocessor compiler. The multiprocessor compiler starts from a description of a real-time stream processing application in a sequential programming language. The *parallelization* phase extracts a task graph and a corresponding CSDF temporal analysis model from this description. The *task analysis* phase provides execution times for the tasks. Given the task graph, the execution times, and the temporal analysis model, the *resource mapping* phase computes a mapping of the tasks to processors. Given this mapping, during the *per processor compilation* phase the tasks are compiled together with the communication libraries and a kernel, such that an executable for our embedded multiprocessor system is obtained.

We obtain an executable for the real-time application that has to execute on our multiprocessor system for which the temporal requirements are satisfied. A real-time application can have temporal requirements in the form of a throughput and a latency constraint. For the execution of such an application on our multiprocessor system, it should be possible to reason and show that the results for the application will be produced on time. We will call a system a *real-time system*, if its correctness does not only depend upon the delivered result, but also upon the moment that the result is delivered. The CSDF analysis model that we use, can be used to guarantee *firm real-time* behavior of an application executed on an embedded multiprocessor system [Moo09, Wig09]. For a firm real-time system, deadline misses are highly undesirable, but not catastrophic.

In the remainder of this section, we will present an overview of our multiprocessor compiler that compiles a stream processing application for our firm real-time system. The phases that we will discuss for our multiprocessor compiler are: the parallelization phase,

the task analysis phase, the resource mapping phase, and the per processor compilation phase.

Parallelization The parallelization phase extracts a task graph and a corresponding CSDF temporal analysis model, from the sequential description of a real-time stream processing application. We introduce a new sequential programming language in which all dependencies can be analyzed. From this language, we can extract a task graph that can be executed on our real-time embedded multiprocessor system. Because we can derive the dependencies, the locations that will be accessed by the tasks are explicit, such that the local memories in the multiprocessor system can be used. Furthermore, our parallelization tool inserts explicit synchronization statements into the tasks, such that we can extract a valid CSDF model. The CSDF model can be used to compute system settings that are required to guarantee the firm real-time behavior.

Task analysis The task analysis phase will compute the execution times of the tasks in the task graph. These execution times will be included in the CSDF model. This model will be used to compute system settings that can be used to guarantee a given throughput constraint. Currently, the task analysis is not implemented for our multiprocessor compiler, such that the user should manually provide the execution times.

In our multiprocessor system, multiple tasks can be executed by a single processor and these tasks can share resources. To guarantee firm real-time behavior, we apply *run-time scheduling* for each shared resource [BMP+04, BMvM07, SBW09, Wig09]. The used run-time schedulers guarantee that a task has a minimum budget for each interval, independent from the other tasks that use the resource. We call these schedulers *budget schedulers*. For a system with budget schedulers, the tasks have a bounded influence on each other via the shared resources.

The task analysis has to extract execution times for the non-blocking parts of the code of a task. These non-blocking parts of a task mainly correspond to the function calls in a task. Because we use budget schedulers in the system, a task has a guaranteed budget per resource that it uses. Therefore, we do not need to consider the execution times of other tasks. Instead, we can derive the execution times for these non-blocking parts of the code of a task in isolation.

Worst-case execution times can be derived, using one of the analysis tools discussed in [WEE+08]. Alternatively, execution times can be measured using an instruction set simulator, as discussed in [BPvM05]. This approach executes the task for a stream of input values and records the total duration of a non-blocking part of the code and the number of memory accesses. The combination of the two results in an estimated upper bound on the execution times for the given input stream. Because we consider a firm real-time application that will not get in an undefined state if a deadline is missed, we can also use these measured upper bounds instead of potentially extremely pessimistic worst-case execution times.

Resource mapping For an application, the resource mapping phase computes the required resource budgets and assigns the required resources in the system. In our multiprocessor system, it is possible to execute multiple applications concurrently and it is possible to enable and disable applications. But, when we allow an application to be enabled, we want to guarantee that this application can be executed, without migrating

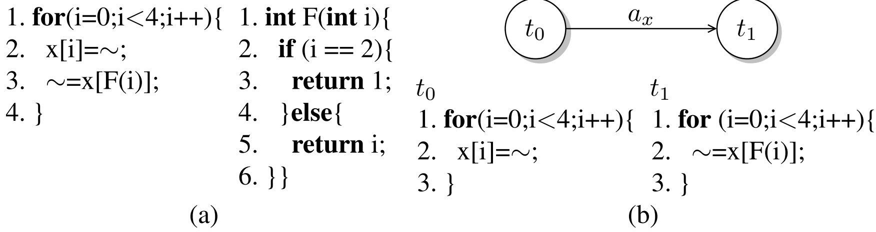


Figure 3.2: (a) An NLP from which (b) a dependency graph is extracted, in which task t_0 and t_1 share dependencies via array a_x

tasks between processors or buffers between memories in the system. Otherwise, we cannot compute a useful estimate of the minimum throughput for the application at compile time. Therefore, we assign the resources in our multiprocessor system at compile time to the applications that will be executed.

For each application a corresponding CSDF model is provided that can be used to compute system settings. A CSDF model that is annotated with the execution times can be used to compute buffer capacities for an application. Currently, the scheduler settings for the tasks of an application have to be provided by the user.

Per processor compilation The final phase in our multiprocessor compiler is the compilation of the tasks. The tasks are linked with a communication library and a kernel that will schedule these tasks on a processor. After compiling the tasks, we obtain an executable for our multiprocessor system.

3.2 Overview of the parallelization phase

In this section, we present the three subphases of the parallelization phase in the multiprocessor compiler. The three subphases are: 1) dependency graph extraction, 2) inter-task communication and synchronization insertion, and 3) temporal analysis model extraction.

The **first phase** extracts a dependency graph. We extract a dependency graph from an NLP by creating a task out of each assignment-statement and by deriving the dependencies between the tasks. The didactic example in Figure 3.2 illustrates the extraction of a task from each assignment-statement. Figure 3.2(b) depicts the dependency graph that we extracted from the NLP in Figure 3.2(a). In this dependency graph, t_0 is the task at the left, t_1 the task at the right, and the array x is called a_x. The arrow in this dependency graph makes explicit that task t_0 writes values into array a_x that will be read by t_1. Note that a dependency graph shows which assignment-statements communicate by reading and writing elements in arrays. For our examples, we use the symbol $\sim$ to represent code that has been omitted for clarity.

The **second phase** inserts inter-task communication and synchronization statements into the tasks. With these statements, the tasks read from and write into buffers. Re-

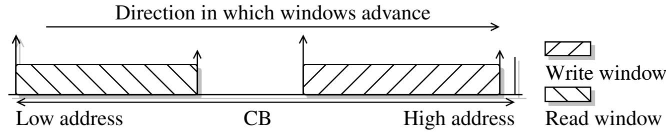


Figure 3.3: A circular buffer with a read window and a write window

placing the accesses of arrays by communication via buffers creates a task graph from the dependency graph. The communication and synchronization statements are inserted using templates. The placement of these statements is such that no race-conditions can occur. Most of the inserted synchronization statements are executed unconditionally, such that we can extract a CSDF analysis model from the inter-task synchronization.

For the inter-task communication, we introduce two new buffer types. For both buffer types, inter-task communication is performed via a so called circular buffer (CB), with a *write window* for each writing task and a *read window* for each reading task, as depicted in Figure 3.3. Each window contains a number of consecutive locations that can be accessed in an arbitrary order by the task, such that non-manifest access patterns inside a window are supported. Each access of a task in its window, is preceded by synchronization to add a location to the window and succeeded by synchronization to remove a location from the window. This synchronization makes that the window advances in the CB. In a CB, the windows can advance independently and therefore the execution of the tasks can be pipelined.

We introduce a CB with *overlapping windows*, for which a written location is removed immediately from the write window after it is written, such that the location can be added to the read windows. Therefore, this buffer can always be used to replace the communication via arrays, even for latency critical cyclic data dependencies. Furthermore, to reduce the synchronization overhead, we introduced a CB with *sliding windows*, for which read windows may not overlap with write windows. However, this buffer cannot always be applied for cyclic data dependencies.

Figure 3.4 depicts the task graph that we created from the dependency graph in Figure 3.2(b). For this example, we replaced the array a_x by CB s_x with overlapping windows. In task t_0, the write access at location i in array a_x has been replaced by **write**(s_x,$\sim$,i) that writes $\sim$ at location i in CB s_x. In t_1, the read access for location $F(i)$ from array a_x has been replaced by **read**(s_x,$F(i)$).

We use an *acquire* call to add locations to a window and a *release* call to remove locations from a window. To make sure that we can access a location in the window, the location has to be acquired first. In Figure 3.4, t_0 has a write window and t_1 has a read window in s_x. The acquire and release statements to use overlapping windows have been inserted into t_0 and t_1. In t_0, at line 1, the acquireS statement, adds 4 locations to the write window, such that all locations of the array a_x are available for writing. The releaseDL statement, at line 4, removes the written location i from the write window, immediately after writing. For t_1 the location $F(i)$ that has to be read is added to the read window with the acquireDL statement at line 2. After executing the for-loop, the

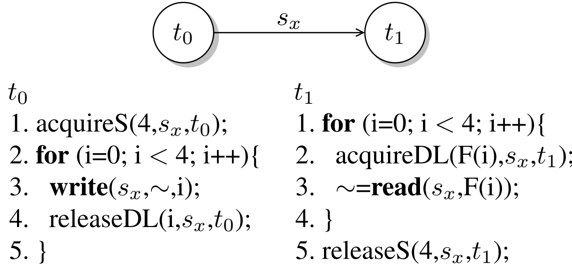


Figure 3.4: Task graph resulting from the dependency graph in Figure 3.2(b)

releaseS statement, at line 5, removes the 4 locations of array a_x from the read window of task t_1. CBs with sliding or overlapping read and write windows and the acquire and release statements to use these windows will be explained in detail in Chapter 5.

In the discussed example, we inserted a CB with overlapping windows, where the windows have the size of the communicated array. Because all locations are available in the window, we can support non-manifest statements. It is possible to use smaller windows if we have manifest statements, but this requires the window size to be computed, as will be presented in Chapter 6. Furthermore, this chapter also presents a template that specifies the placement of synchronization and communication statements in the tasks to use CBs with sliding widows.

In the **third phase** a temporal analysis model is extracted from the inter-task synchronization in the task graph. A temporal analysis model can be used to compute sufficient buffer capacities, such that a given throughput constraint can be satisfied. The approaches presented in [SGB08, WBS07] can be used to compute sufficient initial tokens for a CSDF model to meet a given throughput constraint. The number of initial tokens corresponds to the number of locations required in the buffers. For a buffer, this number of initial tokens states the required buffer capacity, in number of locations.

Figure 3.5 depicts the CSDF model that has been extracted from the task graph in Figure 3.4. In the CSDF model, each node represents an actor v_i. Each actor has a number of consecutive phases for which it is fired. Actor v_0 is annotated with the list $\langle f_0^0, f_0^1, f_0^2, f_0^3 \rangle$ in which each item represents the duration of the firing and the position in the list corresponds to the phase of the actor, so f_0^1 is the firing duration of phase 1 of v_0. Tokens are communicated via queues that are represented by edges in the CSDF model. Each edge is annotated with two lists. The list at the tail of an edge represents the produced tokens and the list at the head of the edge represents the consumed tokens, where the location in the list relates the production or consumption to its phase. An actor can fire if its input queues contain the number of tokens that will be consumed. At the moment an actor fires, it atomically consumes the tokens for the current phase from its input queues. On finishing a firing, an actor atomically produces the tokens for the current phase in its output queues.

In Figure 3.5, actor v_0 models t_0 and actor v_1 models t_1. Each firing of an actor corresponds to an execution of the assignment-statement of the modeled task. CB s_x

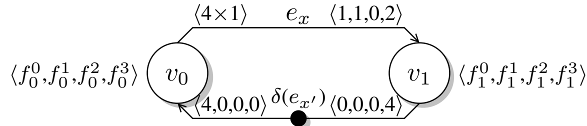


Figure 3.5: The CSDF analysis model extracted from the task graph in Figure 3.4

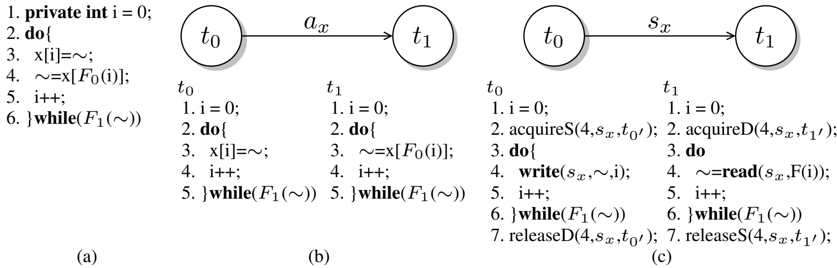


Figure 3.6: (a) An NLP with a non-manifest loop from which (b) a dependency graph is extracted, in which task t_0 and t_1 share dependencies via array a_x and (c) the resulting task graph

is modeled by edge e_x and back-edge $e_{x'}$. The production of tokens models the execution of a release statement that releases locations from a window. The consumption of tokens models the execution of an acquire statement that acquires locations for a window. For the back-edge $e_{x'}$, a number of initial tokens $\delta(e_{x'})$ will be computed such that the throughput will be met. Because these initial tokens correspond to empty locations in the buffer, we can compute the required buffer capacity for s_x. With the algorithm in [WBS07], we computed that for this example a buffer capacity of 4 locations in s_x is sufficient.

We can model the inter-task synchronization via CBs with sliding or overlapping windows in a CSDF model. A task graph can contain hyperedges, because our CB supports multiple producers and consumers. But, the edges in the CSDF model allow only a single producer and consumer. Therefore, in Chapter 7, we will introduce a modeling technique to model the inter-task synchronization via a CB between multiple producers and consumers in the CSDF model.

In addition to the manifest NLP in Figure 3.2 that we used to illustrate the parallelization phases, Figure 3.6 illustrates these phases for a non-manifest NLP. The NLP in Figure 3.6(a) contains a non-manifest loop. The condition of this loop contains the function $F_1(\sim)$ for which the results are undetermined at compile time. In addition, the NLP contains the integer i, which is defined to be private. The keyword private indicates that the assignment-statement at line 5 should not become a separate task, but instead should be copied to each task that reads from the variable i.

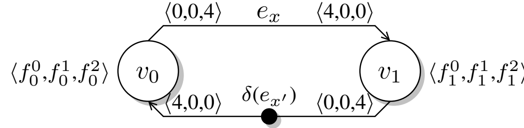


Figure 3.7: The CSDF analysis model extracted from the task graph in Figure 3.4

Figure 3.6(b) illustrates the dependency graph that is extracted from the NLP in Figure 3.6(a). Task t_0 is extracted from the assignment-statement at line 3 and t_1 from the assignment-statement at line 4. Both tasks share a dependency via array a_x.

The task graph in which array a_x is replaced by CB s_x and in which synchronization statements have been inserted into the tasks is depicted in Figure 3.6(c). Synchronization statements have been inserted to use a CB with sliding windows. To handle the non-manifest loop, the tasks t_0 and t_1 initially acquire all locations of a_x in their window by calling the functions acquireS and acquireD, respectively. After executing their while loop the tasks t_0 and t_1 release the locations from their window by calling the functions releaseD and releaseS, respectively.

The temporal analysis model extracted from the task graph in Figure 3.6(c) is shown in Figure 3.7. In this model, actor v_0 models task t_0 and v_1 models t_1. With the algorithm in [WBS07], we computed that for this example a buffer capacity of 4 locations in s_x is sufficient.

3.3 Conclusions

This chapter has presented the internals of our multiprocessor compiler. Our multiprocessor compiler can extract an executable from a real-time stream processing application. For our multiprocessor compiler, we presented four phases. The presented phases are the parallelization phase, the task analysis phase, the resource mapping phase, and the per processor compilation phase. Subsequently, we discussed the parallelization phase in detail, by presenting its three subphases. The subphases for the parallelization are the dependency graph extraction phase, the inter-task communication and synchronization insertion phase, and the temporal analysis model extraction phase.

CHAPTER 4

Dependency graph extraction

Abstract - In this chapter, we present the extraction of a dependency graph from our new sequential programming language. We defined this language, such that we can always extract data dependencies at array granularity. In this language, we require a variable to be assigned a value at most once during an iteration of the endless loop, such that after parallelization only the data dependencies will determine the execution order of the extracted tasks.

In this chapter, we present the extraction of a dependency graph from an NLP described in a sequential programming language. We create a task from each assignment-statement in the NLP. These tasks share dependencies via the arrays that they accesses. In a dependency graph, each node is a task and an edge between multiple nodes corresponds to an array via which these tasks share dependencies. The dependency graph will be used to replace the array accesses of the tasks by communication via buffers. The data dependencies in the dependency graph indicate where synchronization has to be inserted into the code of the tasks.

To extract a dependency graph, we will introduce a new sequential programming language called OIL that enables us to always extract data dependencies between the assignment-statements in our stream processing applications. With the syntax and semantics of OIL, we can express stream processing applications that contain an endless loop and non-manifest statements. New is that our language supports a so called access pattern type for an array, which will provide an optimization directive when the com-

munication via arrays is replaced by communication via buffers. Access pattern types will be discussed in Section 4.3.5. A requirement for an NLP described in OIL is that a variable is assigned a value at most once during an iteration of the endless loop, such that only the data dependencies will determine the execution order of the tasks. Therefore, we can extract function parallelism from an application described in OIL.

The outline of this chapter is as follows. We start this chapter by discussing data dependencies, in Section 4.1. Section 4.2 discusses the data dependencies that we can extract from our sequential programming language OIL. In Section 4.3, we will present the syntax and semantics of OIL. Next, Section 4.4 presents the extraction of a dependency graph from an NLP described in OIL. In case of manifest access patterns, we can derive data dependencies at a finer granularity than the whole array, which will be discussed in Section 4.5. In Section 4.6, the conclusions are presented.

4.1 Data dependencies

In this section, we introduce the different types of data dependencies between statements in an application. These data dependencies have to be maintained after parallelization, to guarantee that the functional behavior of the parallel and sequential application are the same. We will define the so called SA form of data dependencies that enables the parallel execution of statements.

The result from the execution of two assignment-statements that both access different locations in a memory is independent from the execution order of these two assignment-statements. But, if one of the two assignment-statements writes into a location that is read or written by the other assignment-statement, the result of the execution is determined by the execution order of the two assignment-statements. This is called a *data dependency* [ALSU06].

For the assignment-statements from a sequential NLP, the result of the execution of these assignment-statements may depend upon their execution order. In an NLP the execution order of the statements is explicit. The NLP and the in parallel executed assignment-statements will have *functional equivalent behavior*, if for the assignment-statements at least the execution order as indicated by the data dependencies in the NLP are maintained.

Figure 4.1 illustrates the three different data dependencies [ALSU06] that can be identified: a true dependency, an antidependency and an output dependency. There is a *true dependency* between two assignment-statements, if the first assignment-statement writes into a location a and the second reads from the same location. There is an *antidependency* between two assignment-statements, if the first statement reads from a location x and the second writes into x. The *output dependency* relation occurs, if both assignment-statements write into the same location. Note that a location can be a scalar variable or a location in an array.

The antidependency and output dependency relations are referred to as storage related dependencies. Since they are not true dependencies, they can be eliminated by using different locations to store the values. By reducing the number of data dependencies

between assignment-statements, the possible execution orders increase and thereby the possibilities to execute these statements in parallel.

We define an NLP that contains only true dependencies to be in *single assignment (SA)* form [TE68]. This means that during the execution of the NLP, a location is assigned at most once. The SA form is shown in Figure 4.2(a). Note the subtle difference with the *static single assignment (SSA) form* [CFR+91], for which the NLP may contain at most one statement that assigns to a scalar variable x. An NLP in the SSA form is shown in Figure 4.2(b). This example is not in SA form, because location x will be assigned multiple times when the NLP is executed.

To execute the statements from a sequential NLP in parallel, while performing only synchronization for true dependencies, the NLP must be in SA form. If for a location a, the NLP is not in SA form, the statements that access this location a cannot be executed in parallel, because parallel execution will lead to schedule dependent behavior. For the parallel execution, the execution order of the writing statements is unknown. Therefore, after the execution of these statements, the value in a depends on the schedule, such that the functional equivalent behavior between the NLP and the statements that are executed in parallel becomes scheduler dependent. Consider the statements in Figure 4.1. If the two assignment-statements that write into a are executed on different processors, we cannot determine in advance which of the two is the first to write a value into a, such that we do not know which value of a will be read. If for a location the assignments are not in SA form, the read and write accesses must be executed in the sequential order of the NLP, to guarantee functional equivalent behavior.

4.2 Data dependency analysis

Whether it is possible to derive the data dependencies between statements depends on the language in which a program is described. In this section, we will evaluate the influence of specific language constructs on the data dependency analysis. We will motivate why we introduced a new sequential programming language. Furthermore, we will define the data dependencies we can derive from this language.

Languages like C and C++ are often used to program embedded multiprocessor systems. These languages support pointers, where a pointer contains the address of a location in shared memory. For a language that supports pointers, *pointer analysis*, also referred to as pointer alias analysis, is performed to determine at compile time to which memory locations a pointer can point. But, it has been shown that this static pointer anal-

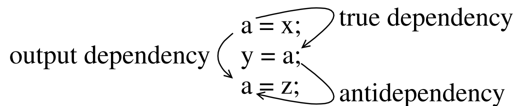


Figure 4.1: Sequential code with a true dependency, an antidependency, and an output dependency

a	b
1. **for**(**int** i=0;i<4;i++){	1. **for**(**int** i=0;i<4;i++){
2. x[i] = i;	2. x = i;
3. }	3. }

Figure 4.2: (a) An NLP in SA form and also in SSA form and (b) an NLP that is in SSA form, but not in SA form

ysis is undecidable in general [Hin01, Hor97, LR92, Lan92, Ram94], amongst others due to non-manifest statements.

Dependency analysis for a language with pointers, has to conservatively insert additional dependencies between statements, if the value of a pointer cannot exactly be determined, to be able to guarantee functional correct behavior. In the extreme case, a pointer can point to any location in the memory, this has two consequences. First of all, dependencies have to be inserted between all assignment-statements in the NLP and the assignment-statement with this pointer. These dependencies require the preceding assignment-statements to have finished their execution before the assignment-statement with this pointer can be executed. Succeeding assignment-statements have to be executed after the execution of this assignment-statement. Secondly, since the pointer can point to *any* location used in the NLP, all data structures have to be statically allocated in a physical shared address space. A local memory that cannot be read by the processor that has to perform the pointer access should not be used. We chose to provide no support for pointers, because one undetermined pointer can already severely restrict the effect of parallelization, by introducing dependencies between all statements.

By not supporting pointers, dependencies can at least be determined at array granularity. We can extract dependencies between statements that access a variable or array with the same name. If for a variable or array we cannot determine the exact dependencies, due to e.g. non-manifest index-expressions, we can always derive an *approximated dependency* between the accesses. The term approximated data dependency was introduced in [BCF97]. We define an approximated dependency between statements that access the same array, as the possibility that these statements will access the same locations in this array. Specifying dependencies by names has the advantage that the dependencies become transparent for the programmer, because he explicitly states the dependencies between the statements at design time.

To extract parallelism from an NLP, it should be in SA form. In general, it is undecidable if an NLP is in SA form. Consider the examples given in Figure 4.3. The example in Figure 4.3(a) illustrates two if-statements that both have a function in their condition, function F_0 and F_1 with the integer x as argument. If these functions are polynomials that have integer variables and coefficients, it is in general undecidable if these functions can return the same result. The solution to Hilbert his tenth problem has shown that such an equality is in general undecidable. Therefore, it is in general undecidable if there is an x for which both conditions are true, such that we cannot determine if the code is in SA form.

```
1. int a;                 1. int a = 1;        1. print(a);
2. int x = input();       2. while(1){
3. if (F0(x)){            3.  if(halt()){
4.   a = 1;               4.    a =a + 1;
5. }                      5.    break;
6. if (F1(x)){            6.  }
7.   a = 2;               7. }
8. }
9. print(a);
        (a)                        (b)
```

Figure 4.3: Two examples that illustrate that the SA form can in general not be determined

The second example in Figure 4.3(b) illustrates the undecidability of a dependency between two statements, according to the theorem given in [Ber66]. Bernstein shows that, in general, exact dependency analysis is impossible, by reducing the dependency analysis between two concurrent statements to the halting problem. The dependency analysis for the while-loop call in Figure 4.3(b) requires us to determine if the condition of the if-statement will ever be true, in which case the code would be in non-SA form. By determining for the condition of the if-statement if it will be true, we would determine if the break for the while-loop will be called, which implies that we solved the halting-problem. Because the halting problem has been shown to be undecidable, we cannot determine if the condition will evaluate to true and therefore we cannot determine if there is only one assignment to a during the execution of the endless loop.

For the subset of affine NLPs with manifest loops, data dependency analysis is possible [Fea91, Fea96, PW94, Pug94]. The approaches in [Fea91, Kie00, VNS07] use this data dependency analysis to determine if an NLP is in SA form and otherwise transforms it into SA form. In [VJBC07] an approach is presented that overcomes the restriction of affine index-expressions. But, none of the previous approaches is suitable for stream processing applications, because neither of them support non-manifest statements.

Because no methods are know to us that transform non-SA code into SA form for stream-processing applications, we start from an application in SA form. This leaves us the choice of starting from a language that only supports expressions to be SA, or a language in which we have to verify SA. The syntax of some applicative languages, like Miranda [Tho95], or SIL [KvMN+92], restricts an application to be in SA form. The syntax can restrict an application to be in SA form, by allowing only recursion to be used, instead of loops. Each recursive function call uses new variables and therefore a variable is assigned once.

We chose not to start from a language in which the syntax restricts us to the usage of recursion, because most stream processing applications are available in a description in which loops are used. Replacing these loops by recursion would require a complete rewrite of the application.

We chose to introduce a new language, called OIL, that supports no pointers and requires the code to be in a kind of SA form. Due to the absence of pointers, we can extract the data dependencies until at least the granularity of arrays. Furthermore, due to the required so called local single assignment (LSA) form, which is a form of SA, no transformation into SA form is required. This enables us to support non-manifest if-statements and loops. The absence of pointers and the LSA form enable us to focus on the important automatic parallelization problems. We have defined OIL, such that we can derive its data dependencies efficiently, we have at least a valid execution order for the statements in an NLP, we can call C-functions from OIL, and we can easily identify the places in the code where synchronization statements should be inserted.

For OIL we accept an NLP in local single assignment (LSA) form, which is a less restrictive form than SA. We define an application to be in *LSA form*, if for one iteration of the endless loop, a variable or location of an array is assigned a value at most once. An NLP may contain initialization statements that precede the endless loop. For the LSA form, we consider the assignments among these initialization statements to be part of the first iteration of the endless loop. We introduce the LSA form, because it is not possible to describe stream processing NLPs in SA form in a language that supports an endless loop. For an NLP with an endless loop, an assignment-statement in the endless loop has an output dependency between its own executions for different iterations of the endless loop, such that the NLP is by definition not in SA form. It is also possible to define a less restrictive form than LSA, as presented in [GBBC11], but this is outside the scope of this thesis.

In an NLP described in OIL, the statements specify a sequential execution order. This execution order is such that the execution of the NLP will not result in deadlock. Without this execution order, the functional behavior of the NLP would be undefined. For the extraction of parallelism, we create tasks from assignment-statements. Furthermore, we will insert synchronization statements to maintain the data dependencies between these assignment-statements. Because the NLP has a valid execution order for the assignment-statements, we can at least execute the tasks in the same order. By using only the data dependencies to determine the execution order, we remove sequence constraints. Removing sequence constraints cannot introduce deadlock, because it cannot introduce cyclic dependencies. In conclusion, the task graph extracted from an NLP described in OIL will not deadlock.

An NLP that is in LSA form and that contains an endless loop, contains output dependencies. An assignment-statement will have an output dependency between its own executions for different iterations of the endless loop. No additional synchronization is required for such an output dependency, because a task cannot start its execution for the next iteration of the endless loop before it finished its execution for the current iteration. This order between iterations is necessary to communicate state between the iterations and to avoid that in our underlying CSDF model the firings of an actor overlap in time.

In OIL, external C-function can be called, such that functions provided by a sequential implementation of the stream processing application can possibly be reused. To keep the data dependencies analyzable, these functions should be *side effect free*. We define a side effect free function as a function that does not use global variables, i.e. it does

not introduce implicit data dependencies. For a side effect free function the result only depends upon the given arguments.

OIL supports calls to external C-functions and describes the dependencies between these function calls. Therefore, OIL can be seen as a coordination language [GC92, Lee07, PA98]. A *pure coordination language* is defined to expresses a top-level description of the dependencies between computations, this in contrast to a pure computation language that only expresses computations. The languages C, C++, Java, and Miranda have a lot of elements from a computation language. In contrast, an NLP has a lot of elements from a coordination language. Because an NLP specifies the dependencies between the function calls, it implicitly specifies the dependency graph. Typically, a coordination language is used on top of a computation language and therefore requires a hierarchical specification of the application. Such a top level coordination language can often be analyzed, which can enable the optimization and parallelization of a given application. In contrast, it can be difficult to perform optimizations and parallelization for an application specified in a single language, because typically it is not possible to analyze the whole application.

A coordination language should not be considered a subset of a computation language. A coordination language can contain constructions related to coordination that are not considered useful in a computation language. Consider for example the access pattern type, which we will present in Section 4.3.5. The access pattern type defines the order in which the elements of an array are accessed. In a coordination language, such an access pattern type provides additional information about the data dependencies and possibly enables optimizations during parallelization, whereas it does not influence the specified functional behavior.

4.3 Omphale input language

In this section, we will discuss the syntax and semantics of OIL. In OIL, an assignment-statement accesses locations in an array. The access pattern of such an assignment-statement in an array depends upon the loops and if-statements that encapsulate this assignment-statement. We will introduce the new concept of an *access type* that specifies the access pattern in an array. An access type can indicate an optimization for the insertion of the buffer, which can be useful, if the compiler cannot derive the access pattern for an array.

The organization of this section is a follows. We will first discuss the syntax and semantics of the assignment-statement, in Section 4.3.1. In Section 4.3.2, the manifest statements that will result in manifest access patterns are discussed. This is extended with the syntax and semantics for stream processing applications, in Section 4.3.3. Subsequently, non-manifest access patterns are discussed, in section 4.3.4. Section 4.3.5 discusses the access types.

4.3.1 The assignment-statement

The statements in an NLP read from and write at locations in arrays. An assignment-statement typically writes a value at a location in an array and can read values from one or more arrays.

In an NLP, the statements access arrays. An array is an ordered list of elements. We identify elements in an array by their location, with location zero the first location and consecutive numbers for the successive locations. Values can be written at and read from the elements in an array. Typically, the location of the element in an array to be accessed is given by an index-expression. Such an index-expression is an expression that can be composed of function calls, variables, and constants. It defines the location that has to be accessed. An index-expression can be non-manifest, but should not evaluate to a location that is not part of the array. Therefore, the array size in number of locations should be declared in an OIL application. Note that we only support arrays with a single dimension.

The execution of an *assignment-statement* assigns values to locations in arrays and can read values from locations in arrays, as illustrated at line 1 and 2 in Figure 4.4. Line 1 in Figure 4.4 depicts our first form of an assignment-statement, with left from the "=" the location to be written and right from the "=" the expression that evaluates to the value that has to be written, this expression may include read accesses in arrays and function calls. For our examples, we use a_x to identify an array x. In our example at line 1, function F_0 computes a value using location zero from array a_y, this value is written at the element of location zero in array a_x. Line 2 depicts an assignment-statement that can write multiple locations. It will write in each array that is prefixed by **out**. In our example, function F_1 will read location one from array a_y and write a computed value at location one from array a_x and at location one from a_z.

1. x[0] = F_0(y[0]);
2. F_1(**out** x[1], **out** z[1], y[1]);
3. F_2(y[2]);

Figure 4.4: Two assignment-statements and a call-statement

A *call-statement* is different from an assignment-statement, because it only reads from arrays. Line 3 in Figure 4.4 illustrates a call-statement that calls function F_2 with location two of array a_y as argument. For the ease of notation, we will address assignment-statements and call-statements as assignment-statements.

The dependencies between assignment-statements from an NLP determine the parallel execution that can be extracted from an NLP. To guarantee correct parallel execution, the data dependencies between the assignment-statements should determine the parallel execution order. The index-expressions used for the array accesses *explicitly* define the data dependencies between assignment-statements, such that synchronization for these dependencies can be inserted.

Implicit dependencies cannot be expressed in OIL, but can be defined inside the functions called from an NLP. Because we cannot analyze implicit dependencies between functions, we require functions to be side effect free.

4.3.2 Manifest access patterns

The execution of an NLP that contains manifest statements results in a manifest access pattern.

Figure 4.5 depicts an NLP that has a manifest access pattern in array a_x. The for-loop encapsulates the if-statement and the if-statement encapsulates the assignment-statement at line 3. This NLP results in a manifest access pattern, because the bounds of the for-loop are constant, and the if-condition and index-expression only depend upon the iterator i_0 for which the values can be determined at compile time.

In OIL we support a manifest loop in the form of a *for-loop*. A for-loop has the syntax **for**($i = b^l; i < b^u; i$+=s), with i the iterator, b^l the lower bound, b^u the upper bound, and s the stride. The lower bound, upper bound, and stride may contain expressions that can even call non-affine functions. The expressions for the bounds and the iterator should be manifest, such that they result in the same sequence of iterations for each execution of the for-loop. The semantic of the for-loop is that for each iteration of the for-loop, the encapsulated statements are executed. The iterations of the for-loop are executed consecutively and may not overlap. Note that we do not require LSA for the iterator of the for-loop, because each task will have its own local iterator. Figure 4.5 illustrates a for-loop at line 1, with an iterator i_0, a lower-bound 0, an upper-bound 5, and a stride of 1.

1. **for**(i_0=0;i_0<5;i_0++){
2.   **if**($F_0(i_0)$){
3.   x[i_0] = $\sim$;
4.   }
5. }

Figure 4.5: A manifest for-loop and if-statement that result in a manifest access pattern

In OIL, manifest if-statements can be used. The syntax of an if-statement is **if**(c), with c the expression for the condition. For a manifest if-statement, we should be able to evaluate the result of the condition at compile time. In the example in Figure 4.5, a function F_0 is used in the condition, with the iterator i_0 of the for-loop as argument. Note that F_0 should be side effect free. The iterator i_0 is independent from input values for the NLP and therefore the sequence of results for the condition of the if-statement can be computed at compile time. Therefore, we can determine when the assignment-statement at line 3 will be executed at compile time.

A constraint for an NLP that has a manifest access pattern is that it should be in LSA form. For an NLP without endless loop, this means that a location in an array is assigned a value at most once. The LSA form is discussed in detail in Section 4.2.

4.3.3 Stream processing

For a stream processing application, an endless loop will be used to consume an endless stream of input values and produce an endless stream of output values. Typically, stream processing applications are not stateless and therefore state has to be communicated between the iterations of the endless loop.

An *endless loop* is depicted in Figure 4.6, with the *while(1)* statement at line 2. This endless loop executes the encapsulated statements an infinite number of times. We do not allow a break statements in the endless loop, because this would change the endless loop into a while-loop with a conditional break. In OIL, we support one endless loop. Supporting multiple endless loops would only results in one loop being executed an infinite number of times.

```
1. x = 1;
2. while(1){
3.   x@=x;
4. }
```

Figure 4.6: An NLP with an endless loop for streaming processing, past iteration communication, and an initialization statement

An NLP with an endless loop can contain *initialization statements*. These statements precede the endless loop and are executed only once. In Figure 4.6, the assignment-statement at line 1 assigns the value one to x, before the endless loop is executed.

Between iterations of an endless loop, state should be stored. For example, for our driver application depicted in Figure 1.1, a sample is decoded or detected, depending upon the state of the previous iteration. We will call the communication of values between iterations of the endless loop, *inter-iteration communication*. A similar concept was introduced for Silage [HRG$^+$90] and Sisal [FCO90] that indicated inter-iteration communication using the @ and *old* operator, respectively.

For inter-iteration communication, we may read the value of an array element for the current and the next iteration, whereas we only allow the value for the array element of the next iteration to be written. At the transition to the next iteration for an endless loop, the value of an array element for the next iteration becomes the value for the array element for the current iteration. Therefore, in the new iteration the value for the array element for the next iteration can be written again. The inter-iteration communication provides FIFO communication between iterations of the endless loop, by transitioning the value for an array element for the next iteration to the value of the array element for the current iteration. Note that there is no iteration transition between the execution of the initial statements and the first iteration of the endless loop. Therefore, only the value for the current iteration of an array should be written by the initialization statements, because the value for the next iteration will be written during the first iteration of the endless loop.

In the example in Figure 4.6, line 3 illustrates the syntax of inter-iteration communication. The element of x in the next iteration is written using the suffix @ and the value of x for the current iteration is read using only x.

We only support inter-iteration communication in FIFO order between iterations of the endless loop. This FIFO communication does not require an index-expression and can therefore be performed between iterations of the endless loop. Furthermore, we know for this FIFO communication that only two copies of the array have to be stored. If we would store a copy of an array for each iteration of the endless loop and allow random access in these copies, potentially for each iteration of the endless loop a new copy of the array has to be stored. We could only execute such an application, if we can determine at compile time the number of copies of an array that have to be stored. Because we cannot compute such a bound for an arbitrary index-expression, we apply FIFO communication over the iterations of the endless loop.

4.3.4 Non-manifest access patterns

A non-manifest access pattern is the result from non-manifest statements in the NLP. We defined non-manifest statements as the combination of non-manifest loops, if-statements, and index-expressions. The access pattern of an assignment-statement in an array can be non-manifest, due to a non-manifest index-expression. The result of a non-manifest index-expression depends upon a value read from an other array. Alternatively, the access pattern of an assignment in an array can be non-manifest, because the assignment-statement is encapsulated by an if-statement or loop with a condition that has a non-manifest expression. In this section, we will discuss the syntax and semantics of the non-manifest if-statement and the non-manifest loop that we support in OIL.

Figure 4.7 depicts three forms of a non-manifest *if-statement*. These if-statements are supported by OIL. Each if-statement has a different syntax, but the same semantics for a language like C. In OIL, the semantics for the if-statements differ. In Figure 4.7(b) and Figure 4.7(c) the if-statements clearly depict mutual exclusive execution of the encapsulated expressions, whereas for Figure 4.7(a) this is undecidable if an arbitrary expression is used for the condition.

(a)	(b)	(c)
1. **if**(F_2(x) <= 0){	1. z=(x==0)**?**F_0(y)**:**F_1(y);	1. **switch**(x){
2. z=F_0(y);		2. **case** 0:
3. }		3. z=F_0(y);
4. **if**(F_3(x) > 0){		4. **default**:
5. z=F_1(y);		5. z=F_1(y);
6. }		6. }

Figure 4.7: Three non-manifest if-statements, (a) an if-statement for which the mutual exclusive execution is in general undecidable, (b) a "?" operator enables mutual exclusive execution between two assignment-statements, and (c) a switch-statement enables mutual exclusive execution between n assignment-statements

For the evaluation of the conditions in Figure 4.7(a), we can in general not decide if either of the encapsulated assignment-statements will be executed. For Figure 4.7(b), the result of the evaluation of the condition will lead to the assignment of either of the two values to z. For the switch-statement in Figure 4.7(c), it is known that only one of the case-statements will be executed.

In OIL, we support a non-manifest loop in the form of a *do-while-loop*, as illustrated in Figure 4.8(a). An alternative that is currently not supported, due to a lack of time, is the *while-loop*, which is depicted in Figure 4.8(b). Both loops show similar behavior. The difference is that the do-while-loop always performs one execution of the statements it encapsulates, whereas the while-loop may never execute the encapsulated statements.

(a)	(b)
1. **do**{	1. z = F_0(y);
2. z = F_0(y);	2. **while**(x == 1){
3. }**while**(x == 1)	3. z = F_0(y);
	4. }

Figure 4.8: (a) A non-manifest do-while-loop that we support in OIL and (b) a while-loop

In OIL, we do not provide special support for parameterized loops and treat them as non-manifest loops. For a parameterized loop, the value of the parameter is unknown at compile-time, but constant during execution. This may enable optimizations. We consider such optimizations as interesting future work.

4.3.5 Access types

In OIL we support access types. An access type is a key word that provides explicit directives for the arrays used in an NLP. We introduce the novel directive of an access pattern type that explicitly specifies the access pattern that the tasks will have in an array. Due to this access pattern type, we can apply inter-task communication and synchronization with less overhead, as we will discuss in Section 6.5. We will also introduce the private access type, which indicates that parallelism should not be extracted from assignment-statements that write this array. The third access type states the number of array elements that will be accessed during an access in an array.

The *access pattern type* should be provided by the user and provides directives about the access pattern of the assignment-statements in an array. Explicit specification is useful, because automatic extraction of such an access pattern can be difficult, due to non-manifest statements. Because an explicitly stated access pattern can be more precise than what we can automatically extract, the resulting code can typically be more efficient. We can identify four classes of access patterns in an array: a FIFO pattern, a static pattern, a bounded non-manifest pattern and an unknown pattern.

For the *FIFO pattern*, only consecutive locations are accessed in the array. This pattern enables us to replace the array by buffers that are suitable for FIFO communication. Such a buffer enables the efficient insertion of synchronization statements, which may

result in a smaller buffer capacity to be sufficient or a higher throughput for the application.

For a *static* access pattern, the access pattern will not change between different iterations of the endless loop and also not between different executions of the NLP. We can derive the static access pattern of an NLP in an array by using symbolic execution, which we will explain in Section 4.4. Typically, such an array can be replaced by a buffer with a capacity that is smaller than the size of the communicated array.

A *bounded non-manifest* access pattern states that the access pattern in the array is not completely unknown. By explicitly stating the variation for the access pattern, it may be possible to insert buffers that can handle the stated worst-case variation in the access pattern.

The fourth access pattern is an *unknown* access pattern in an array. Such an access pattern is non-manifest and therefore we conservatively assume that the locations in an array can be accessed in an arbitrary order.

In OIL, we currently only support the explicit declaration of the FIFO access pattern type. Figure 4.9 depicts an example, where for array a_x the access pattern is defined to be FIFO. Due to the non-manifest loop, we could not have extracted the FIFO access pattern from the NLP. We do not support the explicit declaration of the static and unknown access pattern. For manifest statements, we automatically extract the static access pattern of the assignment-statements in their arrays. For an assignment-statement encapsulated by non-manifest statements, we conservatively assume the unknown access pattern, unless it is explicitly stated for an array that it has a FIFO access pattern.

1. **int fifo** x[5];
2. **private int** i = 0;
3. **do**{
4. x[F_0(i)] = $F_1(\sim)$;
5. i++;
6. }**while**($F_2(\sim)$)

Figure 4.9: An NLP, with the access pattern type *FIFO* defined for array a_x and private access defined for the variable i

The *private access type* for an array explicitly states that parallelism should not be extracted from the assignment-statements that write into this array. Such an access type is necessary to prevent assignment-statements that update a local counter variables, as typically used in the body of a loop, from becoming a separate task. An example of the declaration of a private variable is given at line 2 of Figure 4.9 for the variable i that is used as counter variable in the non-manifest loop. Because a private variable will not be shared between tasks, no LSA is required for this variable.

The *access granularity* states the number of elements that an assignment-statement will access in an array. For an array with more than one element, possibly some of the assignment-statements accesses one array element per execution and other assignment-statements call a function that reads or writes the whole array. For example, by using $x[i]$

an array element is accessed and by using x the whole array will be accessed. In OIL, we currently only support the access of array elements. Support to access the whole array at once is an easy extension that is already partly implemented.

4.4 Dependency graph

We extract parallelism by creating a task from each assignment-statement in an NLP described in OIL. These tasks and their dependencies will be used to extract a dependency graph. In this section, we will first present the formal description of our dependency graph, followed by three examples to illustrate the extraction of a dependency graph from an NLP.

For this section, the organization is as follows. We will explain the extraction of parallelism from an NLP and the resulting dependency graph in Section 4.4.1. In Section 4.4.2, we will give an example of the extraction of a dependency graph from an NLP that has manifest access patterns. An example of the extraction of a dependency graph from an NLP that performs stream processing is presented in Section 4.4.3 and in Section 4.4.4 we give an example of the extraction of a dependency graph from an NLP that has non-manifest access patterns.

4.4.1 Dependency graph extraction

We extract parallelism from an NLP, by creating a task from each assignment-statement that does not assign to a private variable. If the user calls at most one function in each assignment-statement, these functions will be executed in parallel. If the user decides to call multiple functions in a single assignment-statement, these functions will be called from a single task, i.e. these will not be executed in parallel. A task is created from an assignment-statement with its encapsulating control-statements. The term control-statements is used for the if-statements, for-loops, loops, and endless loop that can encapsulate other statements in an NLP. Possibly, the assignment-statement or one of its control-statements reads a private variable. This requires a copy of each assignment-statement that assigns to this private variable to be added to the task. The control-statements that encapsulate these assignment-statements that assign to this private variable should also be added to the task.

We may extract additional parallelism from if-statements and loops. If a function is called in the expression of a condition of an if-statement or a loop, we create a separate task that evaluates this expression. Because such an if-statement or loop can be copied into multiple tasks, creating a separate task for the evaluation of the expression reduces the computational overhead. If the expression does not contain a function call, extracting additional parallelism results in too much overhead. Therefore, the expressions of conditions from control-statements that do not contain a function call will not become tasks.

For an NLP that performs stream processing, the statements preceding the endless loop perform the initialization and are therefore executed only once. For a stream pro-

cessing NLP, these statements will become a single initialization task that is executed once, preceding the execution of all other tasks.

The extraction of parallelism from an NLP results in a directed dependency graph $H = \{T, S, A, \alpha, \rho, \sigma, \theta\}$. For this dependency graph the set of vertices is T. Each vertex $t_i \in T$ represents a task, where the functional behavior of a task is defined by its assignment-statement and its control-statements. The set of arrays is A. The sequential code contains a declaration for each array $a_j \in A$. The set of directed hyperedges is S. A hyperedge $s_j = (\{t_h \mid t_h \in T\}, \{t_i \mid t_i \in T\})$, with $s_j \in S$, is from the tasks in $\{t_h \mid t_h \in T\}$ to the tasks in $\{t_i \mid t_i \in T\}$. Each edge represents a dependency that will be replaced by a buffer. In a buffer s_j, the values of the corresponding array a_j are stored. The size, in number of locations, of the array a_j is given by $\sigma(a_j)$, with $\sigma : A \to \mathbb{N}$. The capacity of buffer s_j is the number of locations $\theta(s_j)$, with $\theta : S \to \mathbb{N}$. For a manifest access pattern in an array a_j, the k-th *access* of task t_i accesses the array location with index $\alpha(t_i, a_j, k)$, with $\alpha : T \times A \times \mathbb{N} \to \mathbb{N}$. For locations and accesses, we start counting at zero. The function $\rho(t_i, a_j)$ returns the total number of accesses performed during one execution of a task t_i that has a manifest access pattern in array a_j, with $\rho : T \times A \to \mathbb{N}$.

In our dependency graph, an edge s_j represents dependencies that are shared between adjacent tasks that all access array a_j. In our dependency graph, we do not illustrate the dependencies between tasks at the granularity of array elements, because the resulting graph would contain too much information, such that the overview would be lost.

Figure 4.10(c), 4.11(b), and 4.12(b) graphically depict a dependency graph. Each node in the graph is annotated with the task it represents and each hyperedge with the array via which the tasks depend upon each other.

4.4.2 Manifest access patterns

In this section, we present an example of the extraction of a dependency graph from an NLP that has manifest access patterns.

Figure 4.10(a) depicts an NLP that has manifest access patterns for array a_x. The NLP contains four assignment-statements that write in and read from array a_x. The assignment-statement at line 9 is non-affine, because in its index-expression it calls the function F_3, which is depicted in Figure 4.10(b). Note that it is also possible that an NLP has a combination of manifest and non-manifest access patterns for an array.

We extract parallelism from the NLP in Figure 4.10(a) by creating a task from each assignment-statement, such that the function calls get distributed over the tasks. This is illustrated in Figure 4.10(c). All tasks share a dependency via the hyperedge a_x.

4.4.3 Stream processing

An example of the extraction of a dependency graph from an NLP that performs stream processing is discussed in this section.

The NLP in Figure 4.11(a) contains an endless loop, such that it can process endless streams of input values. The for-loop at line 2 is an initial statement that will only be

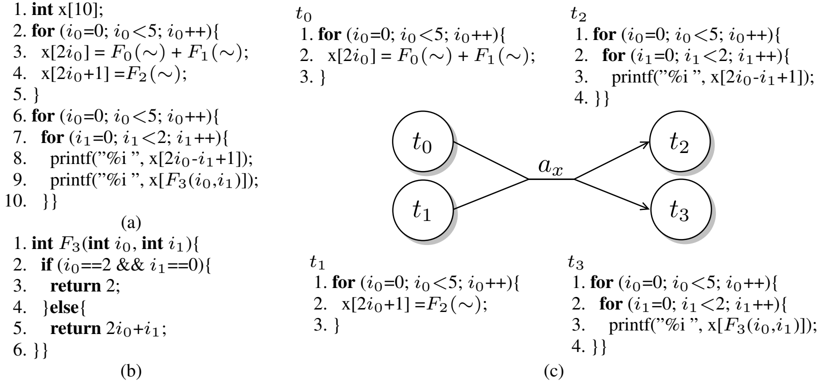


Figure 4.10: (a) An NLP that has manifest access patterns and a non-affine index-expression that calls (b) the function F_3, from which (c) function parallelism is extracted

executed once to write the initial values into array a_y, preceding the execution of the endless loop. Encapsulated in the endless loop, the assignment-statement at line 7 writes into array a_y for the next iteration, with **out** y@[i_0]. Furthermore, this assignment-statement writes into a_x and reads from a_y. The non-manifest if-statement at line 8 executes the assignment-statement at line 9, if $y[i_0]$==1 is true. This results in a non-manifest access patterns in a_x for the assignment-statement at line 9.

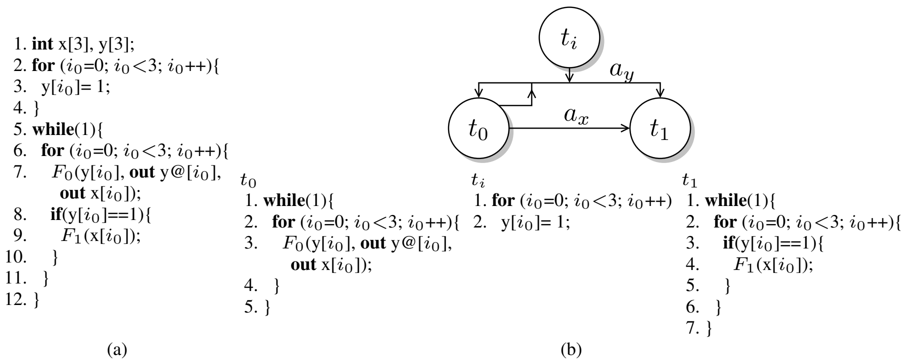


Figure 4.11: (a) An NLP that performs stream processing and (b) the extracted dependency graph

By extracting parallelism from the NLP in Figure 4.11(a), we get the dependency graph that is depicted in Figure 4.11(b). In this dependency graph, task t_0 is created from the assignment-statement at line 7 of the NLP, task t_1 from the assignment-statement at line 9, and the assignment-statement at line 3 becomes the initialization task t_i. Note that we do not create a task to compute the result for the condition of the if-statement,

because this condition does not contain a function call. The dependency graph depicts that there are dependencies between t_0 and t_1 via the edge a_x. Furthermore, there are dependencies between t_0, t_i, and t_1 via a_y. Task t_0 has a cyclic data dependency for a_y, because it reads from and writes into this array.

4.4.4 Non-manifest access patterns

In this section, we will present an example of the extraction of a dependency graph from an NLP that has non-manifest access patterns.

Figure 4.12(a) depicts an NLP described in OIL. The two non-manifest loops in this NLP both call a function in the expression of their condition. Therefore, the tasks t_{c1} and t_{c2} will be extracted from these functions. In this NLP the private variables i and j are used. The loop at line 13 contains only a reading assignment-statement, such that this loop may have an infinite number of iterations.

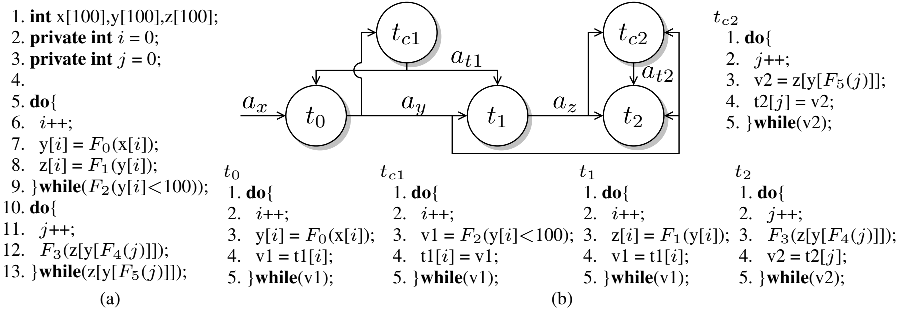


Figure 4.12: (a) An NLP with non-manifest access patterns and (b) the extracted dependency graph

Figure 4.12(b) depicts the dependency graph that is extracted from the NLP in Figure 4.12(a). In the dependency graph, task t_0 contains the assignment-statement from line 7 in the NLP, t_1 the assignment-statement from line 8, t_2 the assignment-statement from line 12, t_{c1} computes the condition from line 9, and t_{c2} computes the condition from line 13. Task t_{c1} writes the results in a_{t1}, such that t_0 and t_1 can use the computed result. Note that there is a latency critical cycle between t_0 and t_{c1}, via a_y and a_{t1}, because both tasks require each others computed values to proceed. Task t_{c2} writes it result in a_{t2}, such that t_2 can use it. Furthermore, the assignment-statements to the private variables i and j at line 6 and 11 do not become separate tasks. These assignment-statements are copied to the extracted tasks that use them, such that the values of these variables are locally computed.

4.5 Access pattern extraction

Initially we approximate data dependencies at array granularity. In this section, we will discuss existing techniques to derive data dependencies, such that the approximated data dependencies can be refined. For the data dependencies that we can derive, we can extract a manifest access pattern. For manifest access patterns, we will distinguish three characteristics, i.e. out-of-order access, multiplicity, and skipping.

For this section, the outline is as follows. In Section 4.5.1, we will discuss two approaches to derive the data dependencies for a task, such that manifest access patterns can be extracted. The three characteristics that we distinguish for manifest access patterns are discussed in Section 4.5.2.

4.5.1 Data dependency derivation

In this section, we discuss two approaches to derive the data dependencies for the accesses in arrays, i.e. static analysis and symbolic execution [VSR96]. *Static analysis* relies on an underlying geometric model, such that transformations for the NLP can be computed, but requires affine expressions. We will apply *symbolic execution*, to derive the data dependencies and extract a manifest access pattern. This approach executes the task, to collect a trace with locations that have been accessed in the arrays.

Our dependency analysis for OIL initially approximates data dependencies at array granularity. If we can derive data dependencies, we can extract a manifest access pattern. A manifest access pattern for a task in an array is defined as an access pattern for which the order of the accessed locations is similar for each execution of the task. We describe an access pattern as the sequence of locations in an array at which a value will be written, or the sequence of locations in an array from which a value will be read. If data dependencies can be derived and a manifest access pattern can be extracted, we may be able to insert the inter-task synchronization more efficiently, which may enable smaller buffers or a higher possible throughput for the application.

Static analysis is an approach to derive the data dependencies by capturing them in the geometric model. This approach captures each index-expression that is used by a task to access locations in an array in the geometric model. Index-expressions can be captured in the geometric model, if they contain affine expression [Fea96, Kie00, PW98]. In the geometric model, the data dependencies can be derived, by using the index-expressions that are used by tasks to access an array. Furthermore, mathematical transformations for the index-expressions can be computed. These transformations can be used to compute optimizations for the tasks, to exploit additional parallelism [WL91] or optimize the data layout for arrays [KKS01, KC02]. However, static analysis requires affine index-expressions before data dependencies can be derived, such that it cannot derive data dependencies for all manifest access patterns.

Symbolic execution can derive data dependencies for all manifest access patterns. Symbolic execution for a task that accesses an array is performed by printing the locations in the array in the order in which the task accesses them. This sequence of accessed locations is the access pattern of the task in this array. For an array that is accessed by

multiple tasks, we compare the access patterns of these tasks in this array and thereby we derive the data dependencies. Symbolic execution has no problem handling non-affine expressions, but requires a manifest access pattern before data dependencies can be derived. Therefore, approaches applying symbolic execution can have difficulties with non-manifest behavior [VSR96].

Because we want to support all manifest access patterns, we will apply symbolic execution to extract the manifest access patterns and to derive the data dependencies. From these manifest access patterns, we derive the locations that will be returned by the function α. For the non-manifest access patterns, we will use the approximated data dependencies.

4.5.2 Access pattern characteristics

For the access pattern of a task in an array, we can identify three characteristics, i.e. out-of-order access, multiplicity, and skipping [TKD04b, HGT07]. These three access pattern characteristics give us the possibility to describe the observed access pattern. Table 4.1 illustrates the access pattern of task t_3 in array a_x from Figure 4.10(c), which we use to explain the three access patterns characteristics.

Table 4.1: Read access pattern of task t_3 from array a_x in Figure 4.10(c)

Access number	0	1	2	3	4	5	6	7	8	9
Location	0	1	2	3	2	5	6	7	8	9

The first access pattern characteristic is *out-of-order access* which we define as the access of non-consecutive locations in an array. In Table 4.10, task t_3 shows an example of out-of-order read accesses in array a_x, by reading location $\alpha(t_3, a_x, 3) = 3$ in access three and the non-consecutive location $\alpha(t_3, a_x, 4) = 2$ during access four. Note that in [TKD04b] the out-of-order access pattern characteristic is defined as a producer and a consumer that have a different access pattern for an array.

The second identified access pattern characteristic is *multiplicity*. Multiplicity occurs if a location in an array is accessed more than once. In Table 4.1, the access pattern of t_3 contains multiplicity, because location two is read for both access two and four in array a_x, so $\alpha(t_3, a_x, 2) = \alpha(t_3, a_x, 4) = 2$ with i_0=1 and i_1=0 for access two and i_0=2 and i_1=0 for access four.

The third access pattern characteristic is *skipping*. Skipping occurs if a location in an array is not accessed. Two types of skipping can be identified: location skipping and value skipping. *Location skipping* occurs if a location is neither written or read. *Value skipping* occurs if a value is written at a location in the array, but this location is never read. The access pattern in Table 4.1 contains skipping for location 4. Depending on the access patterns of the tasks that write into array a_x, it is either location or value skipping.

4.6 Conclusion

In this section, we introduced our sequential programming language OIL from which we can always extract the data dependencies between assignment-statements. OIL does not support pointers and requires the NLP to be in a so called local single assignment form. For this local single assignment form, a variable may be assigned a value at most once during each iteration of the endless loop. Due to these restrictions, we can always extract approximated data dependencies at array granularity. These extracted data dependencies can be used to determine the execution order of the tasks that we will extract, such that we can always extract parallelism.

From an NLP described in OIL, we can extract parallelism, by creating a task from each assignment-statement. We construct a dependency graph, from the data dependencies between the assignment-statements in the tasks. In the following chapters, we will discuss how the array communication, from which these dependencies have been derived, can be replaced by communication via buffers. Initially, all dependencies in the dependency graph are approximated. For tasks with manifest statements, we perform symbolic execution that extracts per task the access patterns in the accessed arrays. By comparing the accesses patterns for an array, the data dependencies between the assignment-statements of the tasks are derived. The manifest access patterns can be used to insert the synchronization statements in such a way that smaller buffers may be sufficient or a higher throughput can be achieved.

CHAPTER 5

Inter-task communication buffers

Abstract - In this chapter, we will present a circular buffer with overlapping windows that can always be used to replace the communication via arrays in the dependency graph. It is shown that this buffer type can be an interesting alternative, because it is not always possible to replace array communication by efficient communication via FIFO buffers, since this may introduce the reordering and buffer selection problems.

In this chapter, we will present two new buffer types that drastically simplify the automatic parallelization of our stream processing applications. Both buffer types apply windows, where a window contains a number of consecutive locations in the buffer. We can choose the size of a window such that an arbitrary access pattern can be performed inside the window, i.e. the access pattern is hidden inside the window. The buffer type with overlapping windows is essential for our approach, because it can always be applied, even for latency critical cyclic data dependencies.

The communication of multiple processors via a shared memory requires a memory consistency model, to define the functional behavior of the application. A memory consistency model defines the order in which other processors observe read and write operations in shared memory. In the dependency graph, we will replace array communication by communication via buffers that rely on a suitable memory consistency model, such that the tasks can be executed on a multiprocessor system.

Often, automatic parallelization approaches apply FIFO buffers for the inter-task communication. But, these FIFO buffers introduce the so called reordering and buffer

selection problems. Efficient solutions for these problems require data dependencies, but have difficulties with approximated data dependencies, as can be encountered in stream processing applications.

We will introduce a new buffer type with *overlapping windows* that is essential for our approach. This buffer applies windows that hide the access patterns. Because these windows can overlap, this buffer type is also applicable for latency critical cycles in the task graph. We will prove that applying this buffer type will not introduce deadlock for the task graph and that the size of the communicated array can be used as buffer capacity. A buffer with overlapping windows is a generalization of a buffer with sliding windows, where the sliding windows are not allowed to overlap. A buffer with sliding windows can be an interesting alternative, because it has a lower synchronization overhead than buffers with overlapping windows. However, this buffer type cannot be applied if cycles are too latency critical. A buffer with sliding windows is a generalization of a FIFO buffer.

The outline of this chapter is as follows. We will first discuss the memory consistency models on which we rely for the implementation of the inter-task communication via shared memory, in Section 5.1. Section 5.2 discusses FIFO buffers and explains the reordering and buffer selection problems that these buffers introduce. To explain the window that can hide approximated data dependencies, we will present a circular buffer (CB) with sliding windows, in Section 5.3. As a generalization of sliding windows, we will introduce a buffer with overlapping windows, in Section 5.4. The conclusions will be presented, in Section 5.5.

5.1 Memory consistency model

The tasks from a dependency graph that are executed on different processors and that communicate via shared memory, require a memory consistency model to prevent the reading of a location before it is written. In contrast to current implementations of synchronization calls [But02, Dij65], we will use four synchronization calls instead of two. Our synchronization calls differentiate between producers and consumers. This avoids the introduction of deadlock due to an incorrect order of inserted synchronization calls in the tasks.

The tasks that communicate by reading from and writing into shared memory are potentially executed in parallel on different processors. For such a system, a memory consistency model is required that defines the order in which write accesses complete [AB09, CGS99]. Write accesses that complete are visible for the processors in the multiprocessor system. Consider the example given in Figure 5.1, taken from [CGS99]. For the execution of the two tasks in this examples, we expect task t_0 to write the value of a in the shared memory and successively set $flag$ to 1, such that t_1 observes that $flag$ is set and successively reads a. The underlying assumption is that for the processor that executes t_1, the write access to a is completed before $flag$ is observed as 1, otherwise the resulting execution will be different from what we expected.

The sequential consistency (SC) model [Lam79] is an example of a strong memory consistency model. This memory consistency model is strong, because it does not allow

t_0	t_1
1. a = 1;	1. **while**(flag == 0);
2. flag = 1;	2. print(a);

Figure 5.1: Two tasks that synchronize via $flag$, from [CGS99]

reordering of shared memory accesses that are issued by a processor, whereas a weaker memory consistency model would allow shared memory accesses from a processor to be reordered. For the SC model, the memory accesses are performed in the order that the statements in the source code specify them. This implies a sequence dependency between each memory access. This is considered a natural match with the expectation of the user, but can prevent the pipelining of memory accesses.

We will assume a multiprocessor system that supports the streaming consistency (StrC) model [BB07]. Compared to the SC model, this is a weak memory consistency model that allows shared memory accesses to be reordered such that they can be pipelined, which potentially results in better performance. To prevent race-conditions between read and write accesses to shared locations in a shared memory, synchronization has to be performed for these accesses. The synchronization should ensure that a shared location is only read after it has been written, otherwise the execution of a consuming task should be stopped (blocked) by a synchronization statement. Furthermore, the synchronization should ensure that a shared location has been read before it is overwritten and otherwise the execution of the producing task should be stopped (blocked). Note that an application written with StrC in mind will behave correctly in a system that implements a stronger memory consistency model that enforces additional ordering constraints for the shared memory accesses.

For the StrC model, synchronization is performed using acquire and release calls. Before accessing a shared location, we perform an *acquire* call for it, this function does not return (blocks) until the location is signaled to be available. Succeeding the access to a shared location, a *release* call signals that the location is available. We differentiate between the acquire for an unwritten and a written location, where the unwritten location is typically acquired as *space* by the producer and the written location as *data* by the consumer. Similarly, a release of *data* by the producer and *space* by the consumer are identified.

For a shared memory location, first an acquire call for space is performed, such that the location can be written. We will call this an acquireSpace call. Successively, a release call for data is performed that signals that the location can be acquired for data. We will call this a releaseData call. A shared location can be acquired for data by a consumer, using an acquireData call. The consumer can release such a location as space using a releaseSpace call, such that a producer is signaled that it can reuse the location by acquiring it for space.

For the StrC model, multiple producers and consumers can perform synchronization via a shared location. The memory consistency model prevents that a read and a write access for a shared location are performed simultaneously. Hence, this does not exclude

simultaneous read accesses nor does it exclude simultaneous write accesses. Therefore, an acquireSpace call for a shared location returns, if the location has been signaled to be available by the releaseData calls of all consumers. Similarly, an acquireData call for a shared location by a consumer returns, if the location has been signaled to be available by the releaseSpace call of all producers. A system should be initialized, such that the first acquireSpace calls of the producers for a location will return. Note that our synchronization via acquire calls and release calls will be based upon the polling of shared variables and not upon using interrupt signals. The synchronization overhead for polling a shared variable can be assigned to the polling task, whereas it is difficult to assign the synchronization overhead caused by interrupts to one of the tasks. Because our system applies budget schedulers [BMP+04, BMvM07, SBW09, Wig09], the polling only uses the budget of the polling task and does not effect the execution time of this task, because the task was already blocked.

Typically, synchronization is performed via locks [But02, Dij65] that provide only an acquire and a release call. In contrast, for our synchronization via shared locations we differentiate between synchronization calls for a producer and a consumer. A producer performs synchronization using the acquireSpace and releaseData calls and a consumer uses the acquireData and releaseSpace calls. Typically, a lock is shared between two or more tasks. Because all acquire calls are equivalent, the first task to acquire a lock depends on the execution order of the tasks on the multiprocessor system. Therefore, the order of inserted acquire calls in the tasks is important to avoid deadlock [But02]. Furthermore, at most one task can acquire a lock at a certain point in time, which may lead to unnecessary blocking of tasks that for example could have acquired the resource concurrently for reading.

Deadlock between two tasks for a lock is illustrated by the example in Figure 5.2(a), for task t_1 and t_2 that both have to acquire the locks a and b before they can start processing. If both t_1 and t_2 perform their first acquire call at the same time, t_1 acquires lock a and t_2 acquires lock b. Consecutively, t_1 has to acquire lock b and t_2 has to acquire lock a, but this is not possible, since both locks are already acquired. Because the acquire calls block, both locks will never be released, such that deadlock occurs. Because there is only a single acquire call, the order in which these acquire calls are inserted into the tasks is important to avoid deadlock. Furthermore, even if the acquire calls are ordered correctly, non-fairness can occur, due to different memory access latencies of the processors for the memory of the lock. If the lock is in the local memory of a processor, the task executed by this processor can reacquire the lock quickly after it released the lock. This may cause unfairness by giving other tasks less change to acquire the lock. In contrast, the example in Figure 5.2(b) uses StrC and therefore the producer calls acquireSpace for buffer a and b and the consumer calls acquireData. In a task, the function *acquireD*, *acquireS*, *releaseD*, or *releaseS* is called to perform an acquireData, acquireSpace, releaseData, or releaseSpace call, respectively. Because the acquireSpace call is successful before the acquireData call, the order of these calls does not matter and will not lead to deadlock.

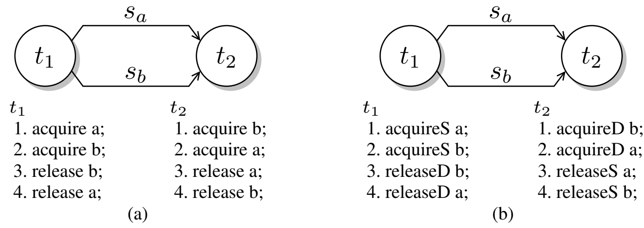


Figure 5.2: (a) Synchronization via a lock that supports a single acquire call, such that deadlock can be introduced, and (b) synchronization according to the StrC model using an acquireData and an acquireSpace call, such that no deadlock is introduced

5.2 FIFO communication issues

FIFO buffers are often used for inter-task communication. To use a FIFO buffer, values have to be written in the order that they will be read. In the dependency graph, the read and write pattern of two tasks in an array can be different. To replace such an array by a FIFO buffer, a reordering task is required, such that the values will be read in the correct order. The extraction of a reordering task and memory leads to the *reordering problem*. Furthermore, our dependency graphs can contain hyperedges, because multiple tasks can read from or write into an array. Since FIFO buffers only support a single reading and writing task, the application of FIFO buffers requires a transformation of the dependency graph, such that at most one task reads from and one task writes into an array. This leads to the so called *buffer selection problem*.

The organization of this section is a follows. We will start by briefly discussing the FIFO buffer, in Section 5.2.1. Subsequently, Section 5.2.2 will discuss the reordering problem for a FIFO buffer that occurs due to unequal read and write patterns. In Section 5.2.3, the buffer selection problem is presented and discussed.

5.2.1 FIFO buffer

Often, inter-task communication is performed via FIFO buffers [CCS+08, CDVS07, DHRA06, RFGEL08, TKD04b, VNS07]. For a FIFO buffer, the first value written into it is the first value that will be read from it and typically the read call is destructive, which means that the read removes the value from the buffer. Therefore, communication via FIFO buffers typically requires local storage if there is read multiplicity, i.e. values will be read more than once.

Typically, a FIFO buffer is implemented with a write and a read call that implicitly perform acquire and release calls. In practice, a FIFO buffer has a finite size and an unread value in the buffer should not be overwritten. Therefore, the write call of a FIFO buffer is typically blocking. This call first performs a blocking acquire, if the acquire returns, a value is written into the FIFO buffer and a release call is performed for

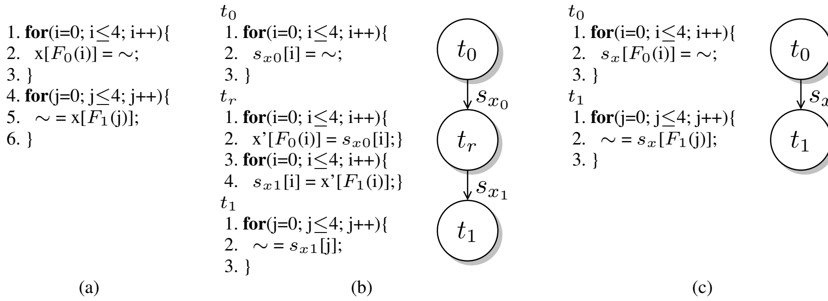


Figure 5.3: (a) Sequential code with two statements that communicate via array a_x from which (b) parallelism is derived with task t_0 writing into the FIFO buffer s_{x0} and t_1 reading from s_{x1} with t_r reordering the values, or alternatively (c) parallelism is derived using a buffer s_x that supports writing at and reading from locations

this location. The read call for a FIFO buffer is typically implemented by performing a blocking acquire operation, succeeded by reading and releasing the location.

The write and read calls for a FIFO buffer cannot be used to write at or read from explicit locations in the buffer. Furthermore, typically only one producer and one consumer communicate via a FIFO buffer. These properties will introduce problems for the inter-task communication. We will discuss these problems in the following sub-sections.

5.2.2 Reordering problem

In this section, the reordering problem is discussed that can occur when communication via arrays is replaced by communication via FIFO buffers. For the tasks in the dependency graph, the array communication should be replaced by inter-task communication via a buffer. The reordering problem occurs, if the array communication is replaced by a FIFO buffer for a producer and a consumer of an array that do not have a similar access pattern in this array. In this case, a reordering task and memory are required, such that the consumer reads values from its FIFO buffer in the same order as they were read from the array initially.

In the dependency graph, tasks write in and read from arrays. If two tasks depend upon each other via an array and their access patterns in the array are the same, the array accesses can be replaced by a FIFO buffer. But, if the access pattern of the reading task contains multiplicity or differs from the access pattern of the writing task, the values written by the producer into the FIFO buffer should be reordered to get the order in which the consumer has to read them. This is what we call the *reordering problem*.

Figure 5.3 illustrates the reordering problem. In Figure 5.3(a), an NLP is depicted in which the assignment-statement at line 2 writes into array a_x and the assignment-statement at line 5 reads from array a_x. Both assignment-statements contain an index-expression in which a function is called to compute the location in the array that has to be accessed.

Figure 5.3(b) illustrates the insertion of two FIFO buffers for inter-task communication. Because FIFO buffers do not support the random access of locations, task t_0 and t_1 access consecutive locations in buffer s_{x0} and s_{x1}, respectively. A reordering task t_r is inserted that reads from FIFO buffer s_{x0} and writes the values reordered into s_{x1}. The depicted reordering task stores the read values in the reordering memory $a_{x'}$. Because t_r cannot write values in s_{x1} before they have been written in $a_{x'}$, first all values are read from s_{x0}. This results in the sequential execution of t_0 and t_1. In [TKD02, TKD04a], an approach is described to perform data dependency analysis between a producer and consumer, such that a reordering task and buffer are inserted that enable more parallel execution. The dependency analysis for this approach requires an NLP with affine index-expressions. Therefore, the functions F_0 and F_1 that are called from the index-expressions would not be supported. However, for the stream processing application domain, we would like to support non-manifest statements.

Figure 5.3(c) illustrates the insertion of a buffer that supports writing at and reading from locations in the buffer, such that no reordering task or memory are required. Such a buffer simplifies automatic parallelization, but requires synchronization that prevents locations from being read before they have been written. We will discuss two buffers that are suitable for approximated data dependencies and that support locations to be written and read using non-manifest index-expressions, in the Sections 5.3 and 5.4.

5.2.3 Buffer selection problem

In this section, the buffer selection problem is discussed for both the consumer and the producer. The buffer selection problem occurs, if the array communication between more than two task in the dependency graph is replaced by inter-task communication via FIFO buffers. Because a FIFO buffer only supports one writing and one reading task, arrays accessed by more than two tasks have to be replaced by multiple FIFO buffers. We identify two types of the buffer selection problem, i.e. the buffer selection problem for the producer and for the consumer. The buffer selection problem occurs for the producer, if multiple tasks may read a value and the producer has to decide into which FIFO buffer a value should be written. The buffer selection problem occurs for the consumer, if values written by multiple tasks have to be read and the consumer has to decide per value from which FIFO buffer it should be read.

Figure 5.4 illustrates the *buffer selection problem for the consumer*. The NLP in Figure 5.4(a) contains two assignment-statements that write into array a_x and one that reads from it. Note that the index-expressions at line 1 and 2 contain a function and that the symbol $\sim$ represents code that has been omitted for clarity, such that these index-expressions can be non-manifest. To exploit the parallelism, we want to extract a task from both the assignment-statements at line 1 and 2 that write into array a_x, rather than keeping them grouped in one task, as depicted in Figure 5.4(b).

The insertion of FIFO buffers for the inter-task communication is depicted in Figure 5.4(c). The writing tasks t_0 and t_1 both write into their own FIFO buffer s_{x0} and s_{x1}. Furthermore, these tasks have to write the location of the communicated value into an additional FIFO buffer $s_{x0'}$ and $s_{x1'}$. The reading task requires a function F_3 that

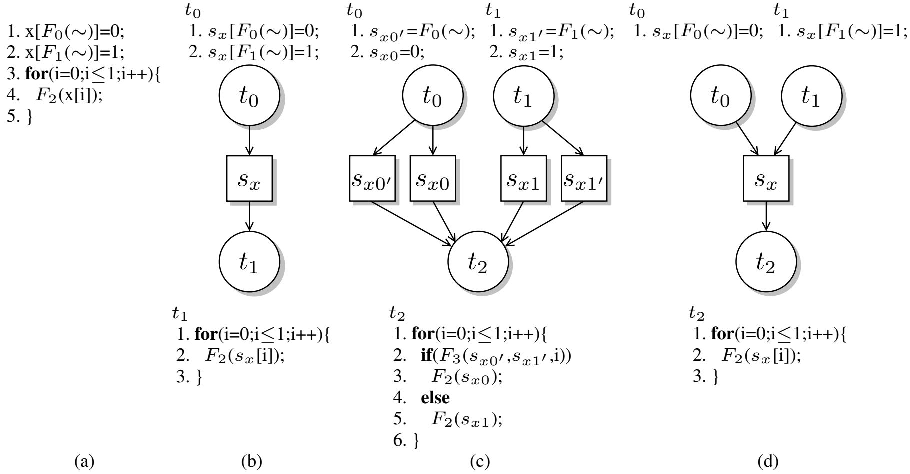


Figure 5.4: (a) Sequential code with multiple assignment-statements that write into a_x from which (b) limited parallelism is derived, due to keeping the writing assignment-statements together, (c) more parallelism is derived because task t_2 reads from the FIFO buffers s_{x0} and s_{x1} using the function F_3, or alternatively (d) parallelism is derived by using a buffer that supports multiple writing tasks

reads the locations of the written values from $s_{x0'}$ and $s_{x1'}$, to select if a value has to be read from FIFO buffer s_{x0} or s_{x1}. A state-of-the-art approach can extract a simple function for F_3, to select the FIFO buffer to be read, only if it can extract data dependencies [Tur07, TKD04b]. This approach requires NLPs with affine index-expressions, such that non-manifest statements are not supported. Due to the manifest access patterns, this approach does not require the buffers $s_{x0'}$ and $s_{x1'}$.

An alternative to inserting FIFO buffers is the insertion of a buffer that supports multiple writing tasks, as depicted in Figure 5.4(d). Such a buffer is desirable, because it simplifies the extraction of function parallelism from our stream processing applications. For such a buffer, no functions for the buffer selection have to be derived and values are written at and read from locations in the buffer. In the remainder of this section, we will explain the buffer selection problem for the producer, followed by a discussion of two new buffer types that allow multiple writing tasks.

Figure 5.5 illustrates the *buffer selection problem for the producer*. Figure 5.5(a) depicts an NLP with one writing and two reading assignment-statements for array a_x. Note that $\sim$ again represents code that is omitted for clarity, such that possibly a non-manifest index-expressions is used to read from a_x.

Figure 5.5(b) depicts the two FIFO buffers s_{x0} and s_{x1} for the inter-task communication that replace a_x. A value written into a FIFO buffer should also be read from it. Therefore, t_0 should only write values into s_{x0} or s_{x1} that will be read by t_1 or t_2, respectively. Task t_1 and t_2 notify t_0 of the locations that they will read, by writing these

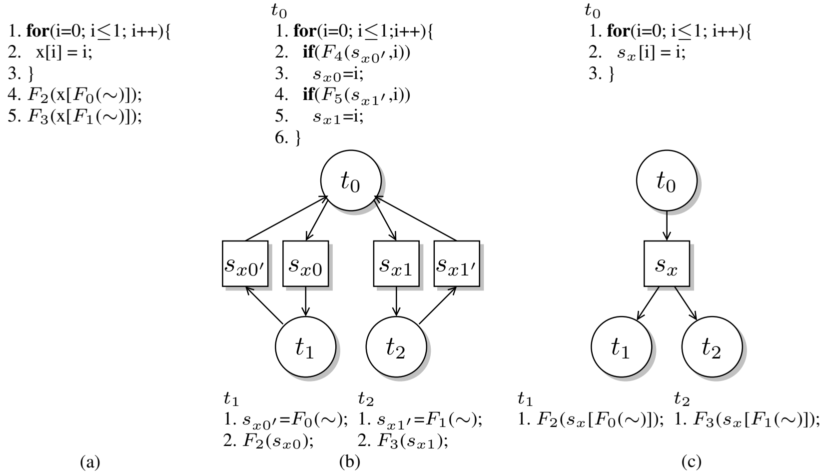


Figure 5.5: (a) Sequential code with multiple assignment-statements that read from array a_x from which (b) parallelism is derived with task t_0 writing into the FIFO buffers s_{x0} and s_{x1} using a condition, or alternatively (c) parallelism is derived using a buffer s_x that supports multiple reading tasks

locations in the buffers $s_{x0'}$ and $s_{x1'}$, respectively. For the producer t_0, an if-statement per FIFO buffer is required with a function in the condition that determines per value if it should be written into the FIFO buffer. For t_0, the functions F_4 and F_5 are used in the conditions to determine if a value should be written into s_{x0} or s_{x1}, respectively. The insertion of the if-statements and the extraction of simple functions form the buffer selection problem for a producer. The approach in [Tur07, TKD04b] can solve this problem for NLPs with affine index-expressions. Their solution does not require $s_{x0'}$ and $s_{x1'}$. However, this approach does not support non-manifest statements.

An alternative to this approach is a buffer that supports multiple reading tasks and in which locations can be read and written, as depicted in Figure 5.5(c). Such a buffer drastically simplifies the derivation of function parallelism for our stream processing applications, because no condition for the writing task has to be derived. Therefore, we will introduce two new buffer types that support multiple reading tasks as well as multiple writing tasks in the Sections 5.3.2 and 5.4.2.

5.3 Buffer with sliding windows

In this section, a CB with multiple sliding read and write windows for the inter-task communication will be presented. Because this buffer supports multiple reading and writing tasks and it supports locations to be accessed, the reordering and buffer selection problems are avoided. We present this CB with sliding windows, to introduce the CB and explain how the windows work. CBs with sliding windows are also presented

in [BBJS08]. In Section 5.4, we will generalize this buffer type by introducing a CB with overlapping windows.

In a CB with sliding windows, a window contains a number of consecutive locations for either reading or writing. The important aspect of a sliding window is that a non-manifest access pattern can be hidden inside a window, such that we can support non-manifest statements. A CB can contain multiple read and write windows, but read windows may not overlap with write windows and vice versa. Read windows may overlap with each other, because a value can be read multiple times. Write windows can overlap without the introduction of race-conditions, because we required our NLP to be in LSA form.

In a CB, a window can slide by performing synchronization calls to add locations to the head of the window and remove locations from the tail of the window. Windows can slide independently from each other, which enables the parallel and pipelined execution of the reading and writing tasks of our stream processing applications. Furthermore, a task only updates the administration for its own window in a CB. Therefore, no atomic read-modify-write operations are required.

A window always contains a number of consecutive locations in the CB. If a location in the middle of the write window is written, it cannot be immediately released from this window, because locations are released from the tail of the window. Therefore, sliding windows may introduce deadlock, if they are used to replace the arrays used in a latency critical cycle of the dependency graph. For such a cycle, it may be required that a value is immediately released from the write window, to make it available for reading.

For this section, the outline is as follows. We will first explain a CB with a sliding read and write window, in Section 5.3.1. Subsequently, the sliding read and write window will be generalized to multiple sliding read and write windows, in Section 5.3.2. Finally, Section 5.3.3 discusses the problem of applying a CB with sliding windows for the inter-task communication on a latency critical cycle in the task graph.

5.3.1 A sliding read and write window

In this section, we will discuss a CB with a sliding read and write window that we will use to replace the array communication in the dependency graph. For sliding windows, care will be taken that the read and write windows do not overlap.

A CB can be implemented with a read pointer r and a write pointer w, as depicted in Figure 5.6. In a CB, a consumer can read the locations between r and w in an arbitrary order and therefore it supports out-of-order access, multiplicity, skipping, and even non-manifest access patterns due to approximated data dependencies. A producer can write the locations between w and r in an arbitrary order. The producer can make the value at location w available to the consumer by increasing w and the consumer can return location r to the producer by increasing r. At the start of the execution of both the producer and the consumer, both r and w point to the same location and w is the first to be increased. The pointers may not overtake each other. When a pointer reaches the end of the CB, it is wrapped around to the beginning.

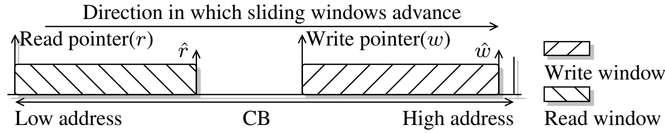


Figure 5.6: A CB with a sliding read window and write window

The reading or writing of an array element l will be replaced by reading or writing the corresponding location l in a CB. For the CB that will be located in shared memory in which it will be accessed by different processors, we will rely on the StrC model, as discussed in Section 5.1.

In a CB, the consumer acquires locations with data between r and w and the producer acquires space between w and r. Starting at r, the consumer acquires a number of consecutive locations with data that can be read, we will call this the read window (RW), as depicted in Figure 5.6. The advantage of acquiring a RW is that only the head of the RW $\hat{r}$ and the tail of the RW, i.e. the read pointer r, have to be stored. This in contrast to storing per location if this location is acquired. In a similar way, a number of consecutive locations between w and r are acquired as a write window (WW), with $\hat{w}$ the head of the WW and w the tail. In the buffer administration, we store the pointers for the RW and WW, such that we know which locations are acquired for data and for space.

Because in a CB a pointer can be increased and wrapped around to the beginning, we will use a *wrap bit* [GNL01, NKG+02] for each pointer. A wrap bit is inverted if its pointer reaches the end of the CB. This wrap bit is necessary in case both windows contain 0 locations and we want to determine if the RW is before or after the WW. If we use a wrap bit per pointer, equal wrap bits indicate that the RW is before the WW and unequal wrap bits indicate that the RW is after the WW. For the pointers r, $\hat{r}$, w, and $\hat{w}$, the wrap bits are indicated by r^b, $\hat{r}^b$, w^b, and $\hat{w}^b$, respectively.

In a CB, an *acquireData* call for the RW increases $\hat{r}$ and an *acquireSpace* call for the WW increases $\hat{w}$. An acquire call is blocking. This means that if an acquireData call is performed for the RW and w is equal to $\hat{r}$, i.e. $w = \hat{r} \wedge w^b = \hat{r}^b$, the acquireData call will not return and also not increase $\hat{r}$, before w has been increased. Note that if w^b and $\hat{r}^b$ are unequal, the WW is before the RW, such that the location can be acquired. An acquireSpace call for the WW blocks, if the location of $\hat{w}$ is equal to r, i.e. $r = \hat{w} \wedge r^b \neq \hat{w}^b$. As for the acquireData call, the acquireSpace call does not increase $\hat{w}$ until it is not blocked anymore. The *releaseSpace* and *releaseData* calls are non-blocking and increase r or w for the RW or WW, respectively.

The producer and consumer can slide their windows independently from each other through the CB, thereby enabling both tasks to be executed in parallel. If the NLP contains an endless loop, the tasks extracted from this loop are executed an infinite number of times. In a CB, the WW is in front of the RW. Therefore, given that the size of the buffer is sufficient, the producer can be a task execution ahead of the consumer. Thus, sliding windows enable the *pipelined execution* of a task graph.

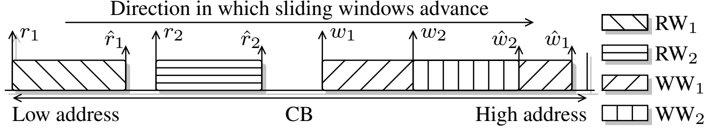


Figure 5.7: A CB with two read window and two write window

For sliding windows, we do not want to depend upon atomic read-modify-write operations like test-and-set and fetch-and-add, because using such operations requires support from the underlying multiprocessor system. We do not require these atomic operations, because w and $\hat{w}$ are only updated by the producer and the consumer only updates r and $\hat{r}$.

5.3.2 Multiple sliding read and write windows

The generalization of a CB with a single WW and RW to a CB with multiple WWs and RWs is discussed in this section. In such a CB, each consumer has its own RW and each producer has its own WW.

Figure 5.7 depicts a CB with two RWs and two WWs. For a RW_n, there is a r_n and a $\hat{r}_n$ and for a WW_m, there is a w_m and a $\hat{w}_m$. Furthermore, each pointer has a wrap bit. The WWs may overlap with each other, because our NLP is in LSA form. Therefore, each location in the WWs will be written by at most one producer, such that no race-conditions will occur. RWs may overlap, because multiple consumers can read the same location without causing race-conditions. For a CB with multiple WWs and RWs, still a WW may not overlap with a RW and vice versa.

For a CB with multiple WWs and RWs, the blocking conditions for the acquireData and acquireSpace calls have to be changed. To ensure that the acquireData call for a RW_n does not cause RW_n to overlap with any of the WWs, the consumer blocks if $\hat{r}_n$ is equal to the tail of one of the WWs. The blocking condition for an acquireData call by a consumer t_n for a CB $s_x = (T_p, T_c)$, with T_p the set of producers, T_c the set of consumers, and $t_n \in T_c$, is:

$$\exists t_p \in T_p(w_p = \hat{r}_n \wedge w_p^b = \hat{r}_n^b) \tag{5.1}$$

Similarly, the acquireSpace call for a WW_m does not return if $\hat{w}_m$ is equal to one of the tails of the RWs. The blocking condition for an acquireSpace call by a producer t_m for a CB $s_x = (T_p, T_c)$, with $t_m \in T_p$, is:

$$\exists t_c \in T_c(r_c = \hat{w}_m \wedge r_c^b \neq \hat{w}_m^b) \tag{5.2}$$

As for a CB with one WW and one RW, the inter-task communication and synchronization via a CB with multiple WWs and RWs does not require atomic read-modify-

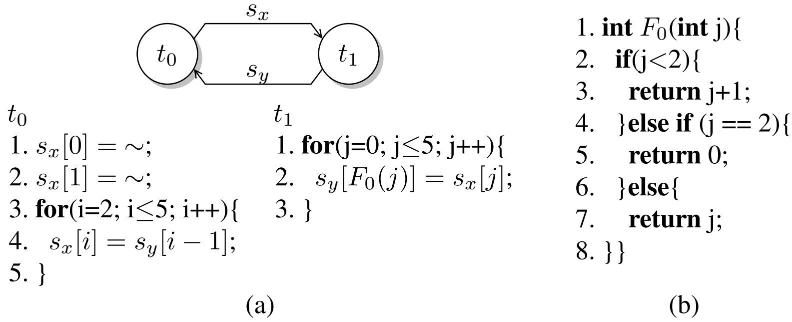


Figure 5.8: (a) A task graph with a cyclic dependency between task t_0 and t_1 and (b) the non-affine function F_0 called by task t_1

write operations. The producer that writes into a WW_n only updates w_n and $\hat{w}_n$, and a consumer that reads from a RW_n only updates r_n and $\hat{r}_n$.

5.3.3 Latency critical cycle problem

For a CB with sliding windows, a value written into a WW is not immediately released from this WW, such that this location cannot be immediately acquired in a RW for reading. Thus, the release of a location from a WW may be delayed for a CB with sliding windows. Therefore, inserting a CB with sliding windows for a latency critical cycle in the dependency graph may introduce deadlock. In this section, we will demonstrate this with an example.

A RW may not overlap with a WW, for a CB with sliding windows. Therefore, these windows are also addressed as non-overlapping windows. For a cyclic dependency graph, the insertion of non-overlapping sliding windows can lead to deadlock in the task graph, which we will demonstrate with the didactic example in Figure 5.8(a). The tasks t_0 and t_1 communicate via the buffers s_x and s_y, according to the sequential code given below these tasks. Task t_1 calls the non-affine function F_0, which is depicted in Figure 5.8(b).

Figure 5.9 depicts the read and write access patterns for the tasks t_0 and t_1 in Figure 5.8(a). The RW and WW in CB s_x and the RW in s_y have one location, because in their access pattern locations are accessed in FIFO order. The WW in s_y requires four locations, due to the irregular access pattern computed by F_0. A location may not be released from this WW before it has been written. Therefore, the locations 1 and 2 in s_y are not released from the WW until location 0 has been written, as depicted in Figure 5.9. Thus, the release of a location from the WW can be delayed, because preceding locations have not yet been written. In this figure, the release of location 1 from the WW in s_y is delayed and therefore depicted in bold. We will now discuss how this delayed release will cause deadlock. To release location 1 from its WW in s_y, task t_1 first has to write location 3 in s_y and read location 3 from s_x. However, before t_0 writes location 3 in s_x and releases it from its WW, it has to read location 1 from s_y. As a consequence, there

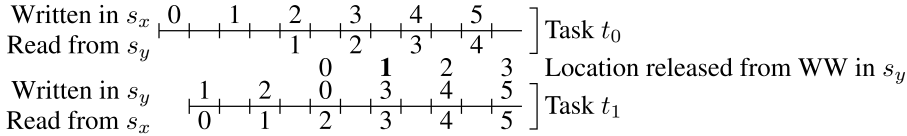


Figure 5.9: Locations read and written by the tasks from Figure 5.8

is a cyclic dependency between t_1 and t_0, where t_1 does not release location 1 from its WW in s_y, because t_0 does not release location 3 from its WW in s_x and vice versa. Because location 1 is not immediately released from the WW in CB s_y after it is written by t_1, the tasks deadlock.

In the following section, a CB with overlapping windows will be discussed, for which a written location is released immediately from a WW, such that it can be acquired for reading. These buffers are suitable to be inserted on latency critical cycles in the dependency graph.

5.4 Buffer with overlapping windows

In this section, we present a CB with multiple overlapping RWs and WWs that can always be used to replace array communication, even for stream processing applications with latency critical cycles. Overlapping windows will not introduce deadlock when used in cyclic task graphs, because a location is released from the WW immediately after it is written. The main advantage of a CB with multiple overlapping windows is that it can always be used to replace the communication via an array, given that a buffer capacity equal to the array size is used. Because the overlapping windows are a generalization of sliding windows, this buffer type also avoids the reordering and buffer selection problems. CBs with overlapping windows are also presented in [BBS09, BBS10].

This section is organized as follows. We will first present the extensions of a sliding RW and WW in a CB towards an overlapping RW and WW, in Section 5.4.1. In Section 5.4.2, the overlapping windows will be generalized to multiple overlapping RWs and WWs. Finally, in Section 5.4.3 we will prove that a CB with overlapping windows can always be applied.

5.4.1 An overlapping read and write window

In this section, we will present the generalization of a sliding WW and RW towards an overlapping WW and RW. The overlapping windows do not introduce deadlock for cyclic task graphs. For overlapping windows, we use a full-bit per location. This full-bit is set immediately after its location has been written, such that the location can be acquired for reading. As for sliding windows, no atomic read-modify-write operations are required to use overlapping windows.

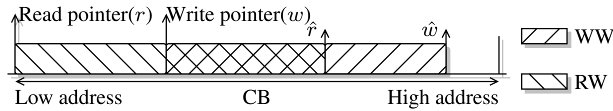


Figure 5.10: A CB with an overlapping RW and WW

As illustrated with the NLP in Figure 5.8, a value should be available for reading directly after it has been written, to prevent deadlock. This requires the producer to release a written location directly from its WW, such that the consumer can acquire it for reading. In contrast, the location at the write pointer w is released from a non-overlapping WW after writing at location k, where w does not necessary equal k. To avoid deadlock, it has to be possible to read the written location k that is possibly after location w, i.e. between w and $\hat{w}$. Therefore, we have to allow reading between w and $\hat{w}$ in the CB. This results in an *overlapping* RW and WW, as depicted in Figure 5.10.

For overlapping windows, per location in the WW it should be administrated if it can be acquired for reading. Therefore, we introduce the *full-bit* that is cleared when its location is acquired for writing and set directly after a value is written at its location. A location in the RW with a set full-bit can be acquired for reading.

A full-bit can either be stored along with its location or in the buffer administration. Some architectures [ABC$^+$95, ACC$^+$90] provide an additional bit for every location in the shared memory that can be used as a full-bit. An alternative is to store full-bits in the buffer administration by using a bit vector, with a full-bit for each location in the CB.

The full-bit is different from the full-empty bit proposed in [CGS99]. A full-bit is only set or cleared by at most one producer. Therefore, no atomic read-modify-write operations are required. In contrast, for a full-empty bit, the producer sets the full-empty bit of a location after writing and a consumer clears the full-empty bit after reading a location for the last time. Possibly, multiple full-empty bits will be stored in a bit vector. In this case, atomic read-modify-write operations are required, to avoid that a producer and a consumer update full-empty bits in the same memory word simultaneously. Otherwise, such a simultaneous update might result in only one of the updates being performed successful.

Overlapping windows use slightly different acquire and release calls compared to sliding windows. The *acquireSpace* call for a WW adds a location to the window by clearing the full-bit of the location consecutive to $\hat{w}$ and acquiring this location by incrementing $\hat{w}$. Note that, as for sliding windows, $\hat{w}$ cannot overtake r and therefore if r is equal to $\hat{w}$, the clearing of the full-bit and the acquire are blocked until r is incremented. After writing a location, a *releaseData* call releases the location from the WW by setting the full-bit of this location. Note that for locations in the buffer that are not written, the full-bit will not be set, because these locations will also not be read.

Before the consumer can read a location, the consumer has to perform an *acquireData* call for it. The acquireData call for a location k blocks, if the location is not between r and $\hat{w}$or if the full-bit of the location is not set. The acquireData call for location k by a consumer t_c in a CB $s_x = (t_p, t_c)$ blocks if the following condition is false:

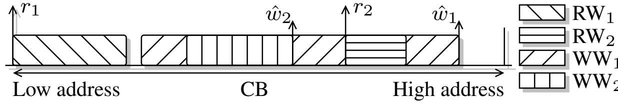


Figure 5.11: A CB with four overlapping windows, two RWs and two WWs

$$((r^b \neq \hat{w}^b \wedge k \mod \theta(s_x) < \hat{w} \wedge k \mod \theta(s_x) \geq r) \vee \\ (r^b = \hat{w}^b \wedge k \mod \theta(s_x) < \hat{w})) \wedge f(k \mod \theta(s_x)) = 1 \tag{5.3}$$

In this equation, the wrap bits ($r^b, \hat{w}^b$) are either equal or unequal. If they are unequal, the window starts at r, wraps around, and ends at $\hat{w}$. Therefore, k should be larger than r or smaller than $\hat{w}$. For equal wrap bits, k is assumed to be larger than r and we verify that k is smaller than $\hat{w}$. Furthermore, the full-bit of the location should be set, so the function $f(k)$ should return true to indicate that the full-bit is set. Note that we use k modulo the capacity of the buffer ($\theta(s_x)$), such that a location larger than the buffer capacity wraps around in the CB.

The consumer can release the location at the tail of its RW with a *releaseSpace* call that increments r.

Updating the read pointer r, write pointer $\hat{w}$, and full-bits requires no atomic read-modify-write operations, as for example test-and-set and fetch-and-add. These operations are not required, because r is only updated by the consumer and $\hat{w}$ and the full-bits only by the producer. Note that due to the full-bits, overlapping windows do not need $\hat{r}$ and w.

5.4.2 Multiple overlapping read and write windows

In this section, we will generalize a CB with an overlapping RW and WW to contain multiple RWs and WWs, such that we avoid the buffer selection problem. As for overlapping windows, multiple overlapping windows do not require atomic read-modify-write operations and can support non-manifest access patterns.

Figure 5.11 depicts a CB with *multiple overlapping RWs and WWs*. Each consumer has a RW$_n$ with a r_n. Multiple RWs can overlap, because a location can be read multiple times among different consumers. Each producer has a WW$_m$ with a $\hat{w}_m$. The WWs of the producers can overlap, because at most one producer will write at a location in the buffer.

For multiple overlapping windows, we do not want to depend upon atomic read-modify-write operations like test-and-set and fetch-and-add, because using such operations require support from the underlying multiprocessor system. Without such operations, a value in the buffer administration can be written and read at the same moment, this results in fair access for the tasks, without unnecessary blocking.

If there are multiple producers in a CB with overlapping windows and only one full-bit per location, this may result in race-conditions. In this case, the first producer to acquire a location for its WW has to clear the full-bit of this location. Possibly, multiple producers acquire such a location simultaneously and therefore each of them will clear the full-bit of this location. For example, the first producer clears the full-bit, immediately writes a value at the location, and sets the full-bit again. The clear operation of one of the other producers can be delayed, such that it clears the full-bit after the first producer has set it. This causes a race-condition, because a consumer possibly never observes the full-bit as being set. This race-condition can be avoided by using an atomic read-modify-write operation for the acquire call of the producer. Alternatively, we avoid such race-conditions, without depending upon atomic read-modify-write operations, by providing each producer its own full-bit for each location in the CB. In this case, there is at most one producer that writes to a full-bit.

For multiple overlapping RWs and WWs, the acquireSpace, acquireData, releaseSpace, releaseData, and write call are slightly changed. In a CB with multiple WWs, a RW can (partly) overtake a WW, to read a value written by another WW. Therefore, a WW should be allowed to overtake such an RW again. But, to prevent unread values from being overwritten, the WW that is ahead of all other WWs may not overtake a RW. Therefore, we change the acquireSpace call. We change the write call, such that it does not write a value into the buffer, if the location to be written is before all read pointers, because this value will never be read.

For multiple overlapping windows, a wrap bit is not sufficient and therefore we generalize it to a wrap counter. For multiple overlapping windows, a read pointer may overtake the head of a write window $\hat{w}$. Consider the case that a $\hat{w}_l$ and $\hat{w}_m$ are equal, but have unequal wrap bits. The wrap bit indicates that one of the WWs is ahead of the other, but we cannot determine if it is WW_l or WW_m. Therefore, we cannot determine which acquireSpace call should block. Note that for sliding windows there would have been a read pointer r_n equal to $\hat{w}_l$ and $\hat{w}_m$, with a wrap bit with the value of the WW that was behind. But for overlapping windows, a read pointer may overtake the head of some write windows. The heads of the WWs and their wrap bits do not provide sufficient information to determine which window is ahead of the other. Therefore, we generalize the wrap bit to a *wrap counter* that starts at zero for each pointer. A read pointer r_n has a wrap counter r_n^c and a $\hat{w}_m$ has a wrap counter $\hat{w}_m^c$. A pointer that reaches the end of a CB is wrapped to the beginning of the CB, its wrap counter is increased by one followed by storing its value modulo three, e.g. increasing $\hat{w}_m^c$ results in $\hat{w}_m^c = (\hat{w}_m^c + 1) \bmod 3$. The wrap counter needs at least three values, from which the pointers will use two values at a time. Therefore, by comparing the counter values of two pointers, we can decide if one of the two is ahead of the other. For efficiency reasons, our implementation of the wrap counter uses four values instead of three, because modulo four can be computed more efficiently by means of bit-masks.

We change the *acquireSpace* call for a producer t_m. The acquire operation clears its own full-bit f_m at location $\hat{w}_m$, followed by increasing $\hat{w}_m$, only if the acquire operation is not blocked. The acquireSpace for a WW_m is blocked, if a read or write pointer is equal to $\hat{w}_m$ and if the wrap counter $\hat{w}_m^c$ is ahead of the wrap counter of this read or

write pointer. The acquireSpace call from a producer t_m in a buffer $s_x = (T_p, T_c)$, with $(t_m \in T_p)$, blocks if the following condition is true:

$$(\exists t_c \in T_c(r_c = \hat{w}_m \wedge ((r_c^c + 1) \bmod 3 = \hat{w}_m^c))\vee \\ (\exists t_p \in T_p(t_p \neq t_m \wedge \hat{w}_p = \hat{w}_m \wedge ((\hat{w}_p^c + 1) \bmod 3 = \hat{w}_m^c)) \tag{5.4}$$

The first line of this equation verifies that the acquireSpace call will not increase $\hat{w}_m$ beyond a read pointer r_c that has a wrap counter r_c^c that is one less than $\hat{w}_m^c$. The second line verifies that no head of a WW is overtaken that has a wrap counter that is one less than $\hat{w}_m^c$. The *releaseData* call of a producer t_m for a location k sets the full-bit f_m that the producer has for the location, i.e. the full-bit f_m at location $k \bmod \theta(s_x)$.

For a consumer, the *acquireData* call for a location is extended to check if one of the producers has written the location. The acquireData call from consumer t_n for location k verifies that there is a producer t_m with a WW_m that is ahead of RW_n, but with k between RW_n and WW_m, and with the full-bit f_m for this location set. Otherwise, the acquireData call blocks. Given a location k and a consumer t_n that reads from a buffer $s_x = (T_p, T_c)$, with $t_n \in T_c$, the acquireData call for location k blocks if the following condition is false:

$$\exists t_p \in T_p(((r_n^c + 1) \bmod 3 = \hat{w}_p^c \wedge (k \bmod \theta(s_x) < \hat{w}_p \vee k \bmod \theta(s_x) \geq r_n))\vee \\ (\hat{w}_p^c = r_n^c \wedge k \bmod \theta(s_x) < \hat{w}_p) \wedge f_p(k \bmod \theta(s_x)) = 1) \tag{5.5}$$

This equation is build up from two parts. The first part, on the first line, of this equation checks for a $\hat{w}_p$ that has been wrapped once more than r_n. In this case k should be smaller than $\hat{w}_p$ or larger than r_n, i.e. the location is between the pointers. The second part considers the wrap counters to be equal and verifies if k is between r_n and $\hat{w}_p$. If one of the two parts is true, the value of the full-bit is checked to be set. The *releaseData* call for a consumer t_n remains unchanged and increases r_n.

Because each producer has its own full-bit per location in the CB, no atomic read-modify-write operations are required for multiple overlapping windows. A tail r_n of a RW_n is only updated by the consumer that reads from this window and a write pointer $\hat{w}_m$ of a WW_m is only updated by the producer that uses this WW for writing. No atomic read-modify-write operations are required for the full-bits, because each producer sets and clears its own full-bits.

5.4.3 Applicability of a CB with overlapping windows

An array can always be replaced by a CB with overlapping windows that has a capacity of at least the size of the array. This section will present a proof that a CB with overlapping windows can always be applied, such that we can always use CBs with overlapping windows for the inter-task communication, even if analysis given a CSDF model fails.

The tasks in the dependency graph read from and write into arrays. This has to be replaced by communication via buffers, to execute the tasks on a multiprocessor system.

As discussed in Section 5.3.3, in some cases replacing the array communication by communication via buffers can introduce deadlock. But for the extracted dependency graph from an NLP described in OIL, we can always replace the array communication by communication via a CB with overlapping windows. Note that such an NLP is in LSA form, such that a written location can be released immediately after writing.

Theorem 5.1. *In a dependency graph, the communication via an array a_x can always be replaced by inter-task communication via a CB s_x with overlapping windows without introducing deadlock, given that the capacity of the CB is at least the size of the communicated array.*

Proof: By definition, an NLP described in OIL from which our task graph is extracted can be executed sequentially. Therefore, there exists a sequential schedule that defines an order in which we can execute the assignment-statements in the NLP. After replacing the arrays by CBs with overlapping windows, in which the CBs have a capacity of at least the size of the arrays, and replacing the assignment-statements in the NLP by tasks in a task graph, we can conclude that these tasks can be executed in the same order. The reason is that values written in these CBs are immediately available for reading. Allowing other execution orders of the tasks is equivalent to the removal of the sequence constraints that enforce the sequential schedule of the tasks. Furthermore, it is known that the task graph is functionally deterministic, because the so called firing rules of the tasks are sequential [LP95]. Removal of sequence constraints from a schedule of a functional deterministic task graph cannot introduce additional cyclic dependencies and therefore cannot introduce deadlock. Because there exists a schedule of the tasks in the task graph and removal of the constraints that enforce this schedule cannot result in deadlock, we conclude that the extracted task graph with overlapping windows is always deadlock-free.
□

It is always possible to use a CB with overlapping windows, if we use the size of the communicated array as buffer capacity. Note that in case of inter-iteration communication an array is used for the current and the next iteration, such that the CB should have a capacity of twice the array size. But, for manifest access patterns we can often use a CB with sliding or overlapping windows that has a buffer capacity that is smaller than the communicated array. First in Chapter 6 we will present how inter-task communication and synchronization statements are inserted for CBs with sliding or overlapping windows. Subsequently, in Chapter 7 we will present the extraction of an analysis model from the synchronization behavior via CBs between the tasks in the task graph, where the analysis model can be used to compute the buffer capacities.

5.5 Conclusion

In this chapter, we discussed suitable memory consistency models that can be used to define the behavior of our tasks, if they are executed in parallel on a multiprocessor system. On top of these memory consistency models, we implement a CB with overlapping windows that can always be used to replace array communication without introducing

deadlock, given that the capacity of the CB is at least the size of the array. This buffer allows multiple producers and consumers. Inside a window, the locations can be accessed in an arbitrary order, such that reordering is not a problem and non-manifest access patterns can be supported.

Preceding the discussion about the CB with overlapping windows, we first discussed FIFO buffers and the buffer selection and reordering problems that the usage of FIFO buffers introduces. A FIFO buffer could be generalized into a CB with sliding windows in which read and write windows are not allowed to overlap. This results in a buffer with a relatively modest synchronization overhead. The CB with sliding windows allows multiple read and write windows, such that it avoids the buffer selection problem. Furthermore, inside a window the locations can be accessed in an arbitrary order, which avoids the reordering problem. A drawback of using CBs with sliding windows is that a location that has been written might not be immediately available for reading. This may introduce deadlock for latency critical cycles in the task graph. Therefore, a CB with multiple overlapping read and write windows has been introduced that releases a written location immediately from the write window after writing.

CHAPTER 6

Communication and synchronization insertion

Abstract - In this chapter, we will present templates that define where communication and synchronization statements must be inserted into the tasks. The insertion of communication and synchronization statements into the tasks transforms the dependency graph into a task graph that can be executed on a multiprocessor system.

This chapter presents the templates that define where communication and synchronization statements must be inserted into the tasks. By inserting these statements, the communication via arrays is replaced by inter-task communication via CBs. Thereby, the dependency graph is transformed into a *task graph* that can be correctly executed on our multiprocessor system. The used templates are defined such that no race-conditions will occur. Furthermore, most of the synchronization statements are executed unconditionally. This makes it possible to extract in many cases a corresponding temporal analysis model in the form of a CSDF graph. The extraction of a CSDF model is the topic of Chapter 7.

First, we will present a generic template that is applicable for OIL applications with non-manifest access patterns. The template defines the placement of statements, to replace array communication by communication via a CB with overlapping windows. To support non-manifest access patterns, we will use windows with the size of the communicated array.

As refinement of the template for non-manifest access patterns, we will present templates that define the placement of communication and synchronization statements for a

manifest access pattern. The first two templates are for CBs with sliding windows and the third template is for CBs with overlapping windows. Typically, for these templates we can use a window in the CB that is smaller than the size of the communicated array. We will present how the size of such a window can be computed. The templates specify the placement of synchronization statements, to use these smaller windows.

We will present two refinements of our generic template, for non-manifest access patterns. First, we will present a refinement for a non-manifest if-statement that contains an assignment-statement that reads from an overlapping window in a CB. In this case a refinement is necessary, to insert the synchronization statements, such that an acquire for the read window is executed unconditionally. The second refinement is the placement of communication and synchronization statements that are needed for the CBs that are read in the expressions of the conditions of the non-manifest loops and if-statements.

To support stream processing applications, we will present a template that defines the placement of communication and synchronization statements into an initialization task. Furthermore, we will present a template, for the placement of these statements for inter-iteration communication.

An NLP can contain an access type that allows the programmer to specify the access pattern of the tasks in an array. Given such an access type, it may be possible to insert synchronization statements into the tasks that immediately precede and succeed the communication statement. This typically results in small windows being used in the CBs. Therefore, we will present a template that defines the placement of synchronization statements, if an access type for an array is specified.

The outline of this chapter is as follows. We will first present a template that defines the placement of communication and synchronization statements for a non-manifest access pattern, in Section 6.1. Section 6.2 refines this template, by presenting a template for the insertion of inter-task communication into a task with a manifest access pattern. The template for non-manifest access patterns is further refined, in Section 6.3, for non-manifest loops and if-statements. A refinement of the templates for stream processing applications is presented, in Section 6.4. Section 6.5 presents a template that defines the placement of synchronization statements given a specified access type by the programmer. Finally, section 6.6 will present the conclusions.

6.1 Generic template

To insert communication and synchronization statements into the tasks of a dependency graph, we will define a template that is in many cases applicable but does potentially not result in the most efficient solution. This template defines the placement of communication and synchronization statements, such that CBs with overlapping windows can be used. The template presented in this section results in windows that contain all locations from the communicated arrays, such that non-manifest index-expressions are supported.

Figure 6.1 depicts the basic template for the placement of inter-task communication and synchronization statements for a CB with overlapping windows into a task. This template contains three phases: the initial phase, the processing phase, and the final phase.

```
 1. int p = 0;
 2. acquireS(σ(a_w),s_w,t_o);              }Initial phase

 3. control{
 4.   p++;
 5.   acquireDL(m_r,s_r,t_o);
 6.   write(s_w, m_w,read(s_r, m_r));       }Processing phase
 7.   releaseDL(m_w,s_w,t_o);
 8. }

 9. p++;                                    }Final phase
10. releaseS(σ(a_r),s_r,t_o);
```

Figure 6.1: A generic template for the placement of synchronization and communication statements for a CB with overlapping windows into a task t_o

During the *initial phase*, for each CB that will be written, the locations of the communicated array are acquired. In the *processing phase*, the assignment-statements that read and write arrays are changed to **read** from and **write** into CBs. The word **control**, which encapsulates the lines 4 until 7, indicates that multiple loops and if-statements can encapsulate the assignment-statements. During the *final phase*, in each read CB, the acquired locations from the communicated array are released.

For our generic template, we acquire all $\sigma(a_w)$ locations of the written array a_w that will be communicated via CB s_w, during the initial phase. This ensures that during the processing phase all locations of a_w will be available for writing. To avoid releasing a location too early, all $\sigma(a_r)$ locations of a read array a_r will be released during the final phase. Note that for overlapping windows, the acquireData and releaseData calls have to immediately precede and succeed the assignment-statement and therefore these calls cannot be performed during the initial or final phase.

Figure 6.1 defines the placement of communication and synchronization statements, to use a CB with *overlapping windows*. We will use shorthand names for the functions acquireSpace, releaseData, acquireData, and releaseSpace. A task t acquires k locations consecutive to the head of the WW in s that contain space, by calling the function *acquireS(k,s,t)*. A task t releases k locations consecutive to the tail of the RW in s that contain space, by calling the function *releaseS(k,s,t)*. The statement to acquire a location m, as given by an index-expression, in s that contains data is *acquireDL(m,s,t)*. To release a location m in s that contains data, the function *releaseDL(m,s,t)* is called. Note that both acquire calls are blocking.

In the template for overlapping windows, read and write accesses of arrays are replaced by read and write statements, to communicate via CBs. A read access of a task t using index-expression m_r in array a_r is replaced by the statement *read(s_r,m_r)* to read the value at location m_r from CB s_r. A write access in array a_w of a task t using index-

expression m_w to write a value v is replaced by *write(s_w,m_w,v)*, to write v at the location m_w in CB s_w.

The template in this section contains a dummy counter p. During the execution of the task, the value of p represents the number of the current so called *synchronization section*. Synchronization sections can be captured in a CSDF model, but this is outside the scope of this chapter. Section 7.3 will discuss synchronization sections in detail.

6.2 Manifest access patterns

In this section, we present two templates that define where communication and synchronization statements must be inserted, in case of manifest access patterns. If a task has a manifest access pattern in an array, typically a read or write window that is smaller than the whole array can be used. This typically results in CBs with a capacity that is smaller than the communicated array. Two templates are needed. In some cases a CB with overlapping windows must be used, to avoid deadlock. But, because a CB with non-overlapping sliding windows has a lower overhead, the usage of these CBs is preferred when deadlock will not be introduced.

We will present the computation of the window size, in number of locations, for a sliding read or write window. We will use a sliding read or write window, such that for each read or write operation of the task, the location to be accessed is acquired. This is possible by encapsulating the reading or writing assignment-statement by an acquire and release statement for at most one location. A nice feature of the presented approach is that a simple expression can be used to determine if an acquire or release statement has to be executed.

A template will be presented that defines the placement of communication and synchronization statements, for a manifest access pattern, such that a CB with sliding windows can be used. In this template, we will use the computed window size for the placement of synchronization statements into a task. Furthermore, a template will be presented that defines the placement of communication and synchronization statements for a CB with overlapping windows, for a manifest access pattern.

For this section, the outline is as follows. First, Section 6.2.1 presents how the size of a sliding window can be computed, given a manifest access pattern. Next, section 6.2.2 presents a template that defines the placement of synchronization statements into a task, to use sliding windows in a CB. A template that defines the placement of synchronization statements into a task to use overlapping windows in a CB is presented, in Section 6.2.3.

6.2.1 Window size computation

We will at most acquire and release one location, for each access in a sliding window. Inside a window, an access pattern can be out-of-order with skipping. However, a window will hide these irregular access patterns. This has the consequence that initially an acquire statement for a number of locations may be required. We will call the number

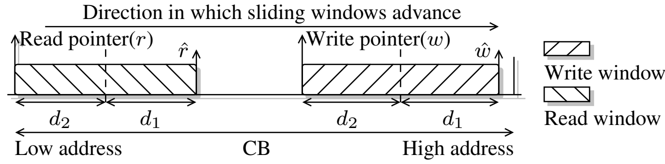


Figure 6.2: A CB with a sliding RW and WW, for which the lead-in (d_1) and lead-out (d_2) are illustrated

of locations that is acquired the *lead-in* (d_1). Furthermore, possibly a number of the accesses should not be followed by a release call, this number is called the *lead-out* (d_2). Together, the lead-in and lead-out determine the size of a sliding window.

We want to insert acquire and release statements into a task, such that the execution of these statements depends upon a simple expression. Therefore, for an accessed window, each assignment-statement in the task that accesses this window is preceded by an acquire statement and succeeded by a release statement for at most one location. Initially, both the head and the tail of the window point to location 0 in the CB. If the assignment-statement that accesses the window is encapsulated by an acquire statement and a release statement for one location, there would be only one location inside the window. For example, just before the fourth execution of such an assignment-statement we would have acquired four locations and released three locations in the CB, such that only location 3 is acquired inside the window. If we need a window with more than one location, initially a number of locations has to be acquired for the window and a number of accesses will not be followed by a release of a location from the window. Figure 6.5 shows a template that defines the placement of acquire and release statements, such that a window with more than one location can be used.

By acquiring a number of locations at the beginning of the execution of a task, a number of locations become available inside the window before the first access. This is the d_1 depicted for the windows in Figure 6.2. Furthermore, by not succeeding the first accesses of the window by releasing a location from this window, a number of locations is kept longer accessible inside the window. This is the d_2 in Figure 6.2. The combination of initially acquired locations and the delayed release of locations determines the window size. We define the *window size* as the maximum distance, in number of locations, between the head and the tail of the window during the execution of the task.

Each access of a task t in a CB s_j is preceded by an acquire operation for at most one location. This is possible until all locations of the communicated array a_j have been acquired. In a CB, the first location to be acquired is location 0. It is possible that the first access of a task accesses location n, with $n > 0$. In this case, preceding the first access and its acquire call, at least n locations should have been acquired. To guarantee that during each access of task t in CB s_j the location to be accessed is acquired, initially a number of locations may have to be acquired. We call this number of locations the *lead-in* $d_1(t, s_j)$, with $d_1 : T \times S \to \mathbb{N}$.

Figure 6.3 depicts the intuition behind the lead-in, for the accesses of t_2 in s_x from the task graph in Figure 4.10(c). In Figure 6.3, the upper sequence lists the indices of the locations acquired by t_2 and the lower sequence list the indices of the read locations. By shifting the sequence with acquired locations to the left, such that no location is acquired after it is accessed, the lead-in is found. Location 1,3,5,7, and 9 are depicted in bold, because they determine the lead-in, which is in this case one, i.e. $d_1(t_2, s_x) = 1$.

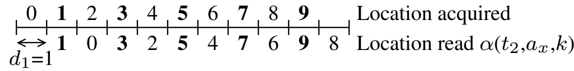


Figure 6.3: The lead-in $d_1(t_2, s_x)$ for task t_2 in CB s_x from Figure 4.10(c)

It is possible to give an expression for the lead-in in a CB. Let t be the considered task, s_j the CB with a_j as corresponding array, and an access counter l for which it holds that $0 \leq l < \rho(t, a_j)$, where $\rho(t, a_j)$ returns the total number of accesses of t in a_j during an iteration of the endless loop and $\alpha(t, a_j, l)$ returns the location that is accessed by t in a_j during access l.

Lemma 6.1. *A lead-in $d_1(t, s_j) = \max_l(\alpha(t, a_j, l) - l)$ is the minimal number of locations acquired before the first access in s_j, such that if each access in s_j is preceded by acquiring one location, it is guaranteed that every location is acquired before it is accessed.*

Proof: Consider an access k, with $0 \leq k < \rho(t, a_j)$, that accesses location $\alpha(t, a_j, k)$. Before this access, at least $\alpha(t, a_j, k) + 1$ locations should be acquired in s_j. Preceding an access in s_j, one location is acquired. Therefore, initially $\alpha(t, a_j, k) + 1 - (k + 1)$ locations should be acquired, if $\alpha(t, a_j, k) > k$. If $\alpha(t, a_j, k) \leq k$, location $\alpha(t, a_j, k)$ is already acquired during access k. To guarantee that for each access the location to be accessed is acquired, the minimal and sufficient number of initial acquired locations $d_1(t, s_j)$ is found by $\max_l(\alpha(t, a_j, l) - l)$, with $0 \leq l < \rho(t, a_j)$. □

To guarantee that during each access of a task in a CB the location to be accessed is still acquired, possibly a number of the initial accesses should not be succeeded by a release call. We will call the number of accesses of a task t in s_j without a release call the *lead-out* $d_2(t, s_j)$, with $d_2 : T \times S \rightarrow \mathbb{N}$.

Figure 6.4 depicts the intuition behind the lead-out, for the reading in s_x by t_3 from Figure 4.10(c). In Figure 6.4, the upper sequence lists the read locations and the lower sequence the released locations. By shifting the sequence with released locations to the right, such that no location is released before it has been read for the last time, the lead-out can be found. Location two is depicted in bold, because it determines the lead-out, which is two in this case, i.e. $d_2(t_3, s_x) = 2$.

We can also give an expression for the lead-out. Let t be the considered task, s_j the CB, and l the access counter for which it holds that $0 \leq l < \rho(t, a_j)$.

Lemma 6.2. *A lead-out $d_2(t, s_j) = \max_l(l - \alpha(t, a_j, l))$, is the minimal number of accesses of t in s_j without a release call, such that succeeding the remaining accesses one*

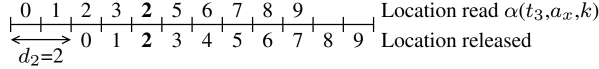


Figure 6.4: The lead-out $d_2(t_3, s_x)$ for t_3 in CB s_x from Figure 4.10(c)

location can be released, without releasing a location before it is accessed for the last time.

Proof: Consider a location with index $\alpha(t, a_j, k)$, with $0 \leq k < \rho(t, a_j)$, that is accessed during the k-th array access. To guarantee that the location with index $\alpha(t, a_j, k)$ is still acquired during access k, at least the first $k - \alpha(t, a_j, k)$ accesses should not be succeeded by releasing one location, if $k \geq \alpha(t, a_j, k)$. In case $k < \alpha(t, a_j, k)$, each access is succeeded by releasing one location. The result will be that the location with index $\alpha(t, a_j, k)$ is released after access k. To make sure that during each access the read or written location is not released yet, at least the first $\max_l(l - \alpha(t, a_j, l))$ accesses should not be succeeded by releasing one location in s_j, with $0 \leq l < \rho(t, a_j)$. □

The lead-out can be negative, in case a consumer skips the first locations in a CB. For example, a consumer t_c that only reads location two from s_j has a lead-out of minus two, i.e. $d_2(t_c, s_j) = -2$. This lead-out is found by applying the expression in Lemma 6.2, with $k = 0$ and $\alpha(t_c, a_j, 0) = 2$.

Together, the lead-in, the lead-out, and one additional location define the size of the window that hides the access pattern. A task t initially acquires $d_1(t, s_j)$ locations and after $d_2(t, s_j)$ accesses, which are only preceded by an acquire call, each access is succeeded by a release call for s_j. For a task t that accesses s_j, the sum of the lead-in $d_1(t, s_j)$, the lead-out $d_2(t, s_j)$, and the location acquired preceding an access, define the number of locations acquired for the window, called the window size $W_s(t, s_j)$, with $W_s : T \times S \to \mathbb{N}$. For a window, at most all elements of the communicated array, as returned by $\sigma(a_j)$, are acquired.

We can give an expression for the window size. Let t be the considered task and s_j the accessed CB.

Theorem 6.1. *For a task t in CB s_j with a lead-in $d_1(t, s_j)$ and a lead-out $d_2(t, s_j)$, a window with window size $W_s(t, s_j) = \min(d_1(t, s_j) + d_2(t, s_j) + 1, \sigma(a_j))$ hides the access pattern.*

Proof: Lemma 6.1 and 6.2 state that the locations to be accessed will be acquired and therefore not yet released. Lemma 6.1 states that a lead-in is determined, such that $d_1(t, s_j) \geq \alpha(t, a_j, l) - l$, this is equal to $\alpha(t, a_j, l) \leq d_1(t, s_j) + l$. Lemma 6.2 states that the lead-out is determined such that $d_2(t, s_j) \geq l - \alpha(t, a_j, l)$, this is equal to $l - d_2(t, s_j) \leq \alpha(t, a_j, l)$. The combination of both ensures that the accessed location is always in the window, $l - d_2(t, s_j) \leq \alpha(t, a_j, l) \leq l + d_1(t, s_j)$. The window size is defined by the difference between the upper bound and the lower bound plus one, i.e. $l + d_1(t, s_j) - (l - d_2(t, s_j)) + 1$. One additional location is required, because the conditional acquire call is performed before we access the CB and the release call after the

access. At most all locations of the array a_j are acquired in CB s_j and therefore the maximum window size is $\sigma(a_j)$. □

6.2.2 Template for sliding windows

This section presents templates that define the placement of communication and synchronization statements into tasks, to communicate via CBs with sliding windows. We present a template for the typical case and a template that also covers the special cases. The inserted inter-task communication can replace array communication with a manifest access pattern. The presented templates are a refinement of the template presented in Section 6.1. For the synchronization of a task, we will distinguish again three phases, namely the initial phase, the processing phase, and the final phase. The initial phase acquires locations for the windows, the processing phase slides the windows, and the final phase releases the remaining locations from the windows.

Figure 6.5 and 6.6 depict templates according to which read and write statements are inserted into a task t. The template in Figure 6.5 defines the placement of statements for the typical case and the template in Figure 6.6 also covers the special cases in which the lead-out for an accessed CB is negative or locations are skipped in an accessed CB. In these templates, t writes into s_w and reads from s_r.

The templates in Figure 6.5 and 6.6 depict three phases: the initial phase, the processing phase, and the final phase. During the *initial phase*, for each accessed CB s_j, lead-in ($d_1(t, s_j)$) locations are acquired, such that during a read or a write access the location to be accessed will be acquired. In the *processing phase*, the assignment-statements that read from and write into arrays are changed to **read** from and **write** into CBs. Furthermore, the assignment-statements are encapsulated by acquire and release statements that will slide the windows in their CBs. The word **control** indicates that multiple manifest loops and if-statements can encapsulate the assignment-statements and their acquire and release statements. In the *final phase*, the remaining locations are released in the accessed CBs.

In addition to the *acquireS* and *releaseS* statements that we already introduced for overlapping windows, we will use the *acquireD* and *releaseD* statements. To release k locations that contain data from the tail of its WW in s, a producer t calls the function *releaseD(k,s,t)*. To acquire the k locations consecutive to the head of its RW in a CB s that contains data, a consumer t calls the function *acquireD(k,s,t)*. Both the acquireS and acquireD calls are blocking.

Among the initial, processing, and final phase, in total $\sigma(a_j)$ locations are acquired and released in a CB s_j, with $\sigma(a_j)$ the number of elements in an array a_j and a_j the array that corresponds to CB s_j. Because each task will acquire and release $\sigma(a_j)$ locations in s_j, all pointers will point to the same location at the end of the execution of these tasks.

In the remainder of this section, the three phases in the templates of Figure 6.5 and 6.6 are presented in detail.

During the **initial phase**, for each accessed CB s_j, $d_1(t, s_j)$ locations are acquired. In the special case that the lead-out is negative for a CB s_j, for which Figure 6.6 shows the template, the for-loop releases $-d_2(t, s_j)$ locations in s_j. The lead-out of a CB can

```
1. int p = 0;                                  }
2. acquireS(ζ(t, s_w),s_w,t);                  } Initial phase
3. acquireD(ζ(t, s_r),s_r,t);                  }
4. int k_r = 0, k_w = 0;                       }

5. control{                                    }
6.   p++;                                      }
7.   if (k_w < σ(a_w) − d_1(t, s_w))           }
8.     acquireS(1,s_w,t);                      }
9.   if (k_r < σ(a_r) − d_1(t, s_r))           }
10.    acquireD(1,s_r,t);                      }
11.  write(s_w, m_w,read(s_r, m_r));           } Processing phase
12.  if (k_w ≥ d_2(t, s_w))                    }
13.    releaseD(1,s_w,t);                      }
14.  if (k_r ≥ d_2(t, s_r))                    }
15.    releaseS(1,s_r,t);                      }
16.  k_r++; k_w++;                             }
17. }                                          }

18. p++;                                       }
19. releaseD(χ(t, s_w),s_w,t);                 } Final phase
20. releaseS(χ(t, s_r),s_r,t);                 }
```

Figure 6.5: A template that defines the placement of communication and synchronization statements into a task t to read from a CB s_r and write into a CB s_w, using sliding windows, if the lead-out is positive for these CBs and no locations are skipped

be negative, if the first locations in an array are skipped. Note that the synchronization for the WWs precedes the synchronization for the RWs. Otherwise, a task that has both a WW and RW in a single CB could deadlock. Deadlock could occur, if the task tries to acquire locations for its RW that this task should have released first from its WW.

The first acquire statement in the initial phase acquires the whole lead-in, if the lead-out is not negative, otherwise it acquires the sum of the locations for the lead-in and the lead-out. The number of acquired locations is given by $\zeta(t, s_j)$, with $\zeta : T \times S \rightarrow \mathbb{N}$ and the function:

$$\zeta(t, s_j) = \begin{cases} d_1(t, s_j) + d_2(t, s_j), \textbf{if } d_2(t, s_j) < 0 \\ d_1(t, s_j), \textbf{otherwise} \end{cases} \tag{6.1}$$

In the template in Figure 6.6, the for-loop in the initial phase acquires and releases $-d_2(t, s_j)$ locations in CB s_j, if the lead-out is negative. Note that this for-loop performs $-d_2(t, s_j)$ acquire and release calls, rather than acquiring $-d_2(t, s_j)$ locations and releasing them. Acquiring and releasing multiple locations increases the window size and potentially the required buffer capacity. This for-loop may acquire and release locations in multiple CBs and therefore the maximum number of locations to be acquired and released among these CBs determines the number of iterations. The number of iterations performed by this for-loop for a task t is returned by the function $\beta(t)$, with $\beta : T \rightarrow \mathbb{N}$ and the function:

$$\beta(t) = \max\{-d_2(t, s) | s = (Z_k, Z_l) \in S \wedge (t \in Z_k \oplus t \in Z_l)\} \tag{6.2}$$

where $\oplus$ represents the exclusive-or operation and Z_k and Z_l are subsets of T, with Z_k representing the producers and Z_l the consumers.

At the end of the initial phase, for each accessed CB s_j, an access counter k_j is introduced that will count the number of accesses of assignment-statements in the CB s_j and will be used during the processing phase. Note that we use a counter per CB, because a task can contain multiple assignment-statements that each may access different CBs. For the templates in Figure 6.5 and 6.6 it would have been sufficient to replace k_r and k_w by a single counter.

The **processing phase**, shown in the templates in Figure 6.5 and 6.6, contains an assignment-statement that accesses CBs and is encapsulated by conditional acquire and release statements. Note that a task may contain multiple assignment-statements and that each of them would be encapsulated by these acquire and release statements. Furthermore, an assignment-statement may be encapsulated by loops or if-statements, but these templates can be applied as long as there is a manifest access pattern for the assignment-statement.

Preceding an assignment-statement that accesses s_w and s_r, a conditional acquireS statement is added for s_w and a conditional acquireD statement is added for s_r. For a CB s_j, using its access counter k_j, this condition checks if there are locations left to acquire, i.e. $k_j < \sigma(a_j) - d_1(t, s_j)$. In the assignment-statement, the read access of element m_r

in an array a_r is replaced by **read**(s_r, m_r), to read location m_r from CB s_r. Similarly, a write access for array a_w at element m_w of value v is replaced by **write**(s_w, m_w, v), to write value v at location m_w in s_w. Succeeding the assignment-statement, conditionally locations are released from the accessed CBs. To determine if a location can be released from s_j, the condition of an if-statement compares the lead-out and the access counter k_j, i.e. $k_j \geq d_2(t, s_j)$. Succeeding the conditional release statements, the access counters of the accessed CBs are incremented.

The last phase is the **final phase**. For this phase, the template in Figure 6.6 contains a for-loop in which acquire and release statements are executed for the remaining elements of the arrays that are communicated via CBs. Due to the skipping access pattern, possibly not all locations in a CB s_j were acquired during the initial and the processing phase of a task t. In the final phase, a for-loop acquires these remaining $\sigma(a_j) - \rho(t, a_j) - d_1(t, s_j)$ locations, where $\rho(t, a_j)$ returns the number of accesses of a task t in an array a_j. In this for-loop, the acquire statements are succeeded by release statements, to slide the window. In both templates, at the end of the final phase, release statements are inserted to release the remaining locations in the accessed CBs.

In the template in Figure 6.6, the number of iterations performed by the for-loop in the final phase of t is given by the function $\eta(t)$. The function $\eta(t)$ returns the maximum number of locations left to be acquired among the accessed CBs, with $\eta : T \to \mathbb{N}$ and the function:

$$\eta(t) = \max\{\sigma(a_j) - \rho(t, a_j) - d_1(t, s_j) | s_j = (Z_k, Z_l) \in S \wedge (t \in Z_k \oplus t \in Z_l)\} \tag{6.3}$$

with $\oplus$ the exclusive-or operation and Z_k and Z_l subsets of T, where Z_k represents the producers and Z_l the consumers.

At the end of the final phase, the remaining locations in the accessed CBs are released, as defined by the templates in Figure 6.5 and 6.6. Possibly, for task t not all locations in a CB s_j have been released, even after executing the for-loop in the final phase in Figure 6.6. If during the iterations of the for-loop no locations were released in s_j, there are $\sigma(a_j) - (\rho(t, a_j) - d_2(t, s_j))$ locations left to be released. If locations have been released during the iterations of the for-loop, then $\sigma(a_j) - \rho(t, a_j) - d_1(t, s_j)$ should be subtracted from this number. For a CB s_j that is accessed by t, the number of locations to be released is $\chi(t, s_j)$, with $\chi : T \times S \to \mathbb{N}$ and the function:

$$\chi(t, s_j) = \begin{cases} \sigma(a_j) - (\rho(t, a_j) - d_2(t, s_j)), \textbf{if } \sigma(a_j) \leq \rho(t, a_j) + d_1(t, s_j) \\ \sigma(a_j) - (\rho(t, a_j) - d_2(t, s_j)) - (\sigma(a_j) - \rho(t, a_j) - d_1(t, s_j)), \textbf{otherwise} \end{cases} \tag{6.4}$$

6.2.3 Template for overlapping windows

This section presents a template that defines the positions at which communication and synchronization statements must be inserted into a task, such that array communication

```
 1. int p = 0;                                              ⎫
 2. acquireS(ζ(t, s_w),s_w,t);                              ⎪
 3. acquireD(ζ(t, s_r),s_r,t);                              ⎪
 4. for (i=0; i<β(t); i++){                                 ⎪
 5.  p++;                                                   ⎪
 6.  if (i > β(t) + d_2(t, s_w)){                           ⎪
 7.    acquireS(1,s_w,t);                                   ⎬ Initial phase
 8.    releaseD(1,s_w,t);}                                  ⎪
 9.  if (i > β(t) + d_2(t, s_r)){                           ⎪
10.    acquireD(1,s_r,t);                                   ⎪
11.    releaseS(1,s_r,t);}                                  ⎪
12. } int k_r = 0, k_w = 0;                                 ⎭

13. control{                                                ⎫
14.   p++;                                                  ⎪
15.  if (k_w < σ(a_w) − d_1(t, s_w))                        ⎪
16.    acquireS(1,s_w,t);                                   ⎪
17.  if (k_r < σ(a_r) − d_1(t, s_r))                        ⎪
18.    acquireD(1,s_r,t);                                   ⎪
19.  write(s_w, m_w,read(s_r, m_r));                        ⎬ Processing phase
20.  if (k_w ≥ d_2(t, s_w))                                 ⎪
21.    releaseD(1,s_w,t);                                   ⎪
22.  if (k_r ≥ d_2(t, s_r))                                 ⎪
23.    releaseS(1,s_r,t);                                   ⎪
24.  k_r++; k_w++;                                          ⎪
25. }                                                       ⎭

25. for (i=0; i<η(t); i++){                                 ⎫
26.  p++;                                                   ⎪
27.  if (i ≤ σ(a_w) − ρ(t, a_w) − d_1(t, s_w)){             ⎪
28.    acquireS(1,s_w,t);                                   ⎪
29.    releaseD(1,s_w,t);}                                  ⎪
30.  if (i ≤ σ(a_r) − ρ(t, a_r) − d_1(t, s_r)){             ⎬ Final phase
31.    acquireD(1,s_r,t);                                   ⎪
32.    releaseS(1,s_r,t);}                                  ⎪
33. }                                                       ⎪
34. p++;                                                    ⎪
35. releaseD(χ(t, s_w),s_w,t);                              ⎪
36. releaseS(χ(t, s_r),s_r,t);                              ⎭
```

Figure 6.6: A template that covers the cases that the lead-out is negative or that locations are skipped, for the insertion of communication and synchronization statements into a task t to read from a CB s_r and write into a s_w, using sliding windows

is replaced by communication via overlapping windows in a CB. The reason that another template is presented is that the statements that must be inserted to use overlapping windows differ slightly from the statements inserted to use sliding windows. The main difference is that in this template a write operation is immediately succeeded by releasing the written location and a read operation is preceded by acquiring the location that has to be read.

Figure 6.7 depicts a template that defines the placement of communication and synchronization statements into a task t, to communicate via CBs with overlapping windows. In this template, task t reads from CB s_r and write into s_w.

As the templates in Figure 6.5 and 6.6, the template in Figure 6.7 contains an initial, a processing, and a final phase. Note that the insertion of acquireS and releaseS statements to use overlapping windows is similar to the statements inserted to use sliding windows. A more compact template in which some acquireS and releaseS statements are grouped is presented in [BBS09], but this template does not illustrate the similarity with the template for sliding windows in Figure 6.6. The main difference between the template in Figure 6.7 and Figure 6.6 is that for overlapping windows the acquireD and releaseD statements are not used, instead acquireDL and releaseDL statements are inserted. These statements are only inserted into the processing phase.

The **initial phase**, as depicted in Figure 6.7, is executed at the beginning of a task. During this phase, for each written CB s_w, lead-in $d_1(t, s_w)$ locations are acquired. For a read CB s_r with a negative lead-out, $-d_2(t, s_r)$ locations will be released from s_r, during the execution of the for-loop. Note that the lead-in of a read CB can be partly acquired by the first acquireS statement and the for-loop, whereas a single acquireS statement for the lead-in would have been sufficient. The depicted insertion of acquireS statements is chosen to illustrate the similarity with the template for sliding windows.

Initially, for a written CB s_w, $\zeta(t, s_w)$ locations are acquired, where the function ζ has been given in Equation 6.1. If the lead-out for s_w is negative, the remaining locations for the lead-in will be acquired in the for-loop. This for-loop also releases locations for a read CB s_r with a negative lead-out. Therefore, the number of iterations performed by this for-loop is the maximum among the negative lead-outs of the accessed CBs, as returned by the function $\beta(t)$, which has already been given for sliding windows in Equation 6.2.

At the end of the initial phase, an access counter is introduced for each accessed CB. During the processing phase, these counters will be used to evaluate if there are locations left to be acquired or released. In Figure 6.7, the access counter k_w is introduced for s_w and k_r for s_r.

During the **processing phase** of a task t, each assignment-statement is preceded by acquire statements and succeeded by release statements, for the accessed CBs. The template of Figure 6.7 depicts that for a written CB conditionally a consecutive location is acquired and that the written location is released. For a read CB, the location to be read is acquired, this verifies that the full-bit of the location is set, and conditionally a consecutive location is released. Succeeding the last release statements, the access counter of each accessed CB is incremented.

Preceding a write access to s_w, an if-statement determines whether there are locations left to acquire, using the access counter k_w, so if $k_w \leq \sigma(t, s_w) - d_1(t, s_w)$. In

1. **int** p = 0;
2. acquireS($\zeta(t, s_w)$,s_w,t);
3. **for** (i=0; i<$\beta(t)$; i++){
4. p++;
5. **if** (i $> \beta(t) + d_2(t, s_w)$)
6. acquireS(1,s_w,t);
7. **if** (i $> \beta(t) + d_2(t, s_r)$)
8. releaseS(1,s_r,t);
9. } **int** k_r = 0, k_w = 0;

Initial phase

10. **control**{
11. p++;
12. **if** ($k_w < \sigma(a_w) - d_1(t, s_w)$)
13. acquireS(1,s_w,t);
14. acquireDL(m_r,s_r,t);
15. **write**(s_w, m_w,**read**(s_r, m_r));
16. releaseDL(m_w,s_w,t);
17. **if** ($k_r \geq d_2(t, s_r)$)
18. releaseS(1,s_r,t);
19. k_r++; k_w++;
20. }

Processing phase

21. **for** (i=0; i<$\eta(t)$; i++){
22. p++;
23. **if** (i $\leq \sigma(a_w) - \rho(t, a_w) - d_1(t, s_w)$)
24. acquireS(1,s_w,t);
25. **if** (i $\leq \sigma(a_r) - \rho(t, a_r) - d_1(t, s_r)$)
26. releaseS(1,s_r,t);
27. }
28. p++;
29. releaseS($\chi(t, s_r)$,s_r,t);

Final phase

Figure 6.7: A template that defines the placement of synchronization and communication statements into a task t to read from the CB s_r and write into s_w, using overlapping windows

the assignment-statement, the write access to array a_w at location m_w is replaced with **write**(m_w,s_w,v), where s_w is the CB corresponding to a_w and v the value to be written. Succeeding an assignment-statement of t that writes location m_w in s_w, location m_w is released by a releaseDL(m_w,s_w,t) statement.

Preceding the assignment-statement that reads location m_r from s_r, in the processing phase of t, an acquireDL(m_r,s_r,t) statement acquires this location. The part of the assignment-statement that reads location m_r from a_r is replaced with **read**(m_r,s_r), to read location m_r from CB s_r. Succeeding the assignment-statement, an if-statement checks if lead-out $d_2(t, s_r)$ accesses have been performed in s_r, by using its access counter k_r, to verify whether a location can be released.

The last phase depicted in Figure 6.7 is the **final phase**. During this phase, a for-loop acquires and releases the remaining locations for the arrays communicated via the CBs. As for the acquireS statements in the initial phase, the releaseS statements could have been grouped into a single release statement for $\sigma(a_r)$-($\rho(t, a_r)$-$d_2(t, s_r)$) locations. We insert a releaseS statement into the for-loop and one succeeding this loop, to keep the similarity with the template for sliding windows.

Due to skipping, possibly not all locations were acquired in a written CB s_w. In the for-loop of the final phase, acquire calls are performed for the remaining $\sigma(a_w)$-$\rho(t, a_w)$-$d_1(t, s_w)$ locations in s_w. The function $\rho(t, a_w)$ returns for one execution of t the number of accesses in a_w. For a read CB s_r, there can be remaining locations that have to be released, due to multiplicity, skipping, or out-of-order access. In the for-loop of the final phase, release calls are performed for $\sigma(a_r)$-$\rho(t, a_r)$-$d_1(t, s_r)$ locations in s_r.

The number of iterations for the for-loop during the final phase is determined by the number of locations that have to be acquired among the written CBs and released among the read CBs. As for sliding windows, the number of iterations is given by the function $\eta(t)$ in Equation 6.3.

Succeeding the for-loop, a release statement is inserted to release the remaining $\chi(t, s_r)$ locations, with the function χ given in Equation 6.4.

6.3 Non-manifest access patterns

In the dependency graph, a data dependency between tasks may be approximated. The exact data dependency may not be known at compile time, due to a non-manifest loop, if-statement, or index-expression. In Section 6.1, the template has already been presented that defines the insertion of statements for the inter-task communication via CBs, given a non-manifest access pattern. In this section, we will present two extensions of this template. The first extension considers the case that an assignment-statement is encapsulated by a non-manifest if-statement. If this assignment-statement uses an overlapping RW, additional synchronization statements have to be inserted. For the second extension, we will replace the arrays that are accessed in the expressions of the conditions of the non-manifest loops and if-statements by communication and synchronization statements for CBs.

The organization of this section is a follows. We start by discussing how we can handle a non-manifest if-statement, in Section 6.3.1. Next, we will present templates to insert synchronization statements for the expressions of conditions of non-manifest loops and if-statements, in Section 6.3.2.

6.3.1 Handling if-statements

In this section, we will present a new type of acquire statement that has to be inserted in the body of a non-manifest if-statement. This acquire statement is necessary if the assignment-statement reads from an overlapping RW. For such an assignment-statement, the acquireDL statement is executed conditionally, because it is inserted in the body of the if-statement. The acquireDL statement immediately precedes the assignment-statement. But, the releaseS call will be executed unconditionally, as defined in the template in Figure 6.1. As a consequence, it can occur that in an endless loop only the release statement is executed. The result can be that the buffer gets in an undefined state. To prevent this, an else part will be added to every non-manifest if-statement in which a overlapping RW is accessed. In this else part, an acquire statement is executed. The result is that, independent of the condition of the non-manifest if-statement, always an acquire statement will be executed.

We have to guarantee that during each access of a task t in its window in a CB s_x the location to be accessed is acquired. Therefore, as presented in Sections 6.1, we acquire all locations of the communicated array a_x in the window during the initial phase and release these locations from the window during the final phase, if the task has a non-manifest access pattern in a_x. Thus, if it is possible we will perform unconditional synchronization. Only, for a CB with overlapping windows, we execute the acquireDL and releaseDL statement conditionally, because they have to immediately precede and succeed the assignment-statement.

If a CB with overlapping windows is applied for inter-task communication for an assignment-statement that is executed in the body of non-manifest loops or if-statements, the *acquireDL* or *releaseDL* statements are executed conditionally. The releaseDL statement can be conditionally executed, because it should only set the full-bit for a location if data has been written to it. But the conditional execution of the acquireDL statement for a CB s_x implies that the task t is possibly not blocked for s_x. Therefore, task t can unconditionally perform the corresponding releaseS statements in its final phase. This can be problematic, if the task performs stream processing. Potentially, executing unconditional releaseS statements for s_x may increase r_x beyond all heads of the WWs, resulting in an undefined state for the buffer.

For a task t_n with an endless loop that communicates via overlapping windows, a conditionally executed acquireDL statement is possibly never executed, such that releaseS statements can increment r_n unconditionally and without blocking. Because the read pointers and the pointers to the heads of the WWs point to locations in the CB and the wrap counters are stored modulo 3, no logical locations are stored in a CB. Therefore, for a task with an endless loop that performs non-blocking and unconditional releaseS calls, it becomes undetermined which locations have been released from the RW.

(a)	(b)
1. **while**(1){	1. **int** o_x =0;
2. **if** ($\sim$){	2. **while**(1){
3. F_0(x[y]);	3. **if** ($\sim$){
4. }	4. acquireDL(1 + o_x,s_x,t_n);
5. }	5. F_0(**read**(s_x,y + o_x));
	6. } **else**{
	7. acquireV(s_x,t_n);
	8. }
	9. releaseS($\sigma(a_x)$,s_x,t_n);
	10. $o_x = (o_x + \sigma(a_x)) \mod \theta(s_x)$;
	11. }

Figure 6.8: (a) A task t_n with a non-manifest if-statement that conditionally executes the function F_0 and (b) the template that defines the placement of the synchronization statements for the CB s_x with overlapping windows

Only an assignment-statement that is in the body of a non-manifest if-statement, can result in an acquireDL statement that is never executed. A non-manifest loop is implemented by a do-while-loop, such that we always execute at least one iteration of this loop. Therefore, for a task that reads from a CB with overlapping windows in a non-manifest loop, during each iteration of its endless loop, at least one acquireDL statement will be executed, such that the RW cannot overtake the WWs.

For a non-manifest if-statement that contains an assignment-statement that accesses an overlapping RW in CB s_x, we add an *else-statement* in which we will execute a so called acquireV statement. This else-statement with the acquireV call makes the execution of an acquire for s_x unconditional. Figure 6.8(a) depicts a task with an endless loop and a non-manifest if-statement with in its body an assignment-statement that reads from array a_x. Figure 6.8(b) defines the synchronization statements that have to be inserted to replace the communication via array a_x by a CB s_x with overlapping windows. The acquireDL statement for s_x precedes the assignment-statement that reads from s_x. An else part is added to the if-statement, with at line 7 a blocking *acquireV* statement for s_x, such that the conditional synchronization for s_x becomes unconditional.

We introduce the *acquireV* statement, to verify for a task t_n in a CB s_x that its RW$_n$ is not ahead of the WWs. Otherwise, the acquireV call will block the execution of the task, until a WW is ahead of its RW$_n$. If for each iteration of the endless loop the acquireV statement verifies that r_n is not ahead of the WWs, during the final phase all locations from RW$_n$ can be released, given that the buffer capacity is at least the array size. Therefore, after the releases in the final phase, RW$_n$ can be at most the array size ($\sigma(s_x)$) in numbers of locations ahead of the WWs. In this case, the wrap counter r_n^c can at most indicate that it is wrapped once more than the pointers to head of the WWs. This will not block a producer, because it can still determine if a read pointer is ahead or before the head of the WW.

An *acquireV* call should block the consumer, if its RW is ahead of all WWs in a CB with overlapping windows. The acquireV statement of a consumer t_n verifies that there is a producer t_m with a WW_m that is ahead of its RW_n. Otherwise, the acquireV call will not return. For a consumer t_n that reads from a buffer $s_x = (T_p, T_c)$, with $t_c \in T_c$, an acquireV call will not return as long as the following equation is false:

$$\exists t_p \in T_p(((r_n^c + 1) \bmod 3 = \hat{w}_p^c) \vee (r_n^c = \hat{w}_p^c \wedge r_n \leq \hat{w}_p)) \tag{6.5}$$

For a consumer t_n, this equation contains two parts. The first part verifies that $\hat{w}_p$ has been wrapped once more than r_n, in this case the acquireV call will not block. The second part checks if the wrap counters r_n^c and $\hat{w}_p^c$ are equal and verifies that $\hat{w}_p$ is larger or equal to r_n. Note that the wrap counters can have three values and because at most two values will be used, for two pointers we can determine if one pointer is ahead of the other.

6.3.2 Handling expressions in conditions

An expression of a condition of a non-manifest loop or if-statement can read from arrays. In this case, communication and synchronization statements must be inserted, to replace the communication via arrays in these expressions by inter-task communication via CBs. In this section, a template is presented that defines the placement of these statements.

A non-manifest loop or if-statement contains a condition with an expression. Initially, arrays are read in this expression. These read accesses in arrays have to be replaced by communication via CBs. Array accesses can be performed in the expression of the condition. But, we cannot insert the communication and synchronization statements to read from CBs into this expression. Therefore, we introduce an assignment-statement that writes the result of the expression into a scalar variable c. This assignment-statement can be encapsulated by synchronization calls. The expression for the condition is replaced by c.

The template in Figure 6.9 contains a non-manifest if-statement and the template in Figure 6.10 a non-manifest loop. In both templates, the expression of the condition is evaluated in a separate assignment-statement that writes the result of the expression into a scalar variable c. In Figure 6.9(b), t_0 stores the result of the condition in c, such that this assignment-statement can be encapsulated by acquire and release statements. In the if-statement at line 6, the expression is replaced by the variable c.

If the expression for a condition contains a function call, the result for the condition will be computed in a separate task. In the dependency graph, this task writes the result of the expression into an array a_c. In this case, the assignment-statement that we create from the expression of the condition of an if-statement or a loop, can read the result from array a_c. For the task graph, a_c will be replaced by a CB s_c. Both the template in Figure 6.9(b) and Figure 6.10(b) depict a task t_c that computes the result for the condition and communicates it via a CB s_c.

Figure 6.9(b) depicts the template to insert synchronization statements into the two tasks t_c and $t_{0'}$. Theses tasks have been extracted from the task t_0 in Figure 6.9(a) that

```
t0                  tc                               t0'
1. control{         1. int kc = 0;                   1. int t, kc = 0;
2.  if((F0(~)){     2. control{                      2. control{
3.    ~;            3.  acquireS(1,sc,tc);           3.  acquireD(1,sc,t0');
4.  }               4.  write(sc,kc++, F0(~));       4.  c = read(sc,kc++);
5. }                5.  releaseD(1,sc,tc);           5.  releaseS(1,sc,t0');
                    6. }                             6.  if(c){
                                                     7.    ~;
                                                     8.  }
                                                     9. }
        (a)                              (b)
```

Figure 6.9: (a) A task t_0 with an if-statement with the function F_0(x) called from the expression of its condition and (b) the template that defines the task t_c that is extracted to compute the result of the expression of the condition, which it communicates via s_c, such that $t_{0'}$ can read it and execute the body of the if-statement

contains a function call in the condition of its if-statement. Note that t_c computes a function $F_0(\sim)$, where $\sim$ could require CBs to be read. The insertion of synchronization statements for these CBs depends upon the access pattern of this task in these CBs.

In Figure 6.9(b), task t_c computes the result for the condition and writes it into CB s_c. Potentially, an if-statement is executed multiple times, because it is contained in a loop. In this case, both t_c and $t_{0'}$ would be encapsulated by this loop and t_c would write multiple results for the condition into s_c that will be read by $t_{0'}$. The write pattern in s_c should not contain multiplicity to avoid results from being overwritten before they have been read. Therefore, each result for the condition should be written at a new location in s_c. Because the number of executions of the body of the if-statement can be undetermined, the number of results written into a_c and s_c is unknown. However, it is known that t_c will write a result that will be read by $t_{0'}$, such that FIFO communication can be inserted for the computed result.

Task t_c and $t_{0'}$ perform FIFO communication via s_c. We will perform this FIFO communication via a CB with a sliding read and write window that both contain one location. Therefore, the acquire and release statements for s_c immediately precede and succeed the assignment-statement at line 4 for t_c and $t_{0'}$. This synchronization can be conditional, because both tasks execute the synchronization in the same loops and therefore will execute it the same number of times. In a CB, we have to access a location. Therefore, in both tasks the counter k_c is introduced for the location that the task will access in s_c. This counter is initialized at 0 and incremented succeeding each access in s_c. As for the synchronization, the tasks t_c and $t_{0'}$ will increment k_c in the same loop nest, such that each location written into s_c will be read. Note that if the capacity of s_c is larger than 1, the execution of t_c and $t_{0'}$ may be pipelined, because t_c can compute a number of values ahead.

(a) t_0

```
1. control{
2.   do{
3.     ~;
4.   }while(F0(~))
5. }
```

(b) t_c

```
1. int c, kc = 0;
2. control{
3.   do{
4.     c = F0(~);
5.     acquireS(1,sc,tc);
6.     write(sc,kc++,c);
7.     releaseD(1,sc,tc);
8.   }while(c)
9. }
```

$t_{0'}$

```
1. int c, kc = 0;
2. control{
3.   do{
4.     ~;
5.     acquireD(1,sc,t0');
6.     c = read(sc,kc++);
7.     releaseS(1,sc,t0');
8.   }while(c)
9. }
```

Figure 6.10: (a) A task t_0 with a do-while-loop with the function F_0 that is called in the expression for its condition and (b) the template that defines the task t_c that computes the condition and communicates the result via s_c, such that $t_{0'}$ can use it in the condition of its do-while-loop

The template in Figure 6.9(b) depicts only $t_{0'}$ that reads the result for its condition from s_c. But if n tasks would have been extracted from the body of the if-statement in Figure 6.9(a), each of them could have read the result from s_c, because this buffer supports multiple reading tasks.

In Section 4.3.4 three different forms of an if-statement have been discussed, i.e. an if-statement, a "?" operator, and a switch-statement. The semantics of these different forms gave an indication for the data dependencies between statements and the LSA form of the code. In the task graph, only the functional behavior is important and therefore these forms are replaced by an if-statement, as depicted in Figure 6.9.

Figure 6.10(b) depicts a template to insert synchronization statements into t_c and $t_{0'}$ that are extracted from the do-while-loop in Figure 6.10(a). A task t_c is extracted to compute the result for the expression of the condition. Task t_c communicates the results of the condition in FIFO order via s_c, where both $t_{0'}$ and t_c use a counter k_c to access locations in s_c. Because both tasks encapsulate the synchronization with the same loop nest, the synchronization will be performed the same number of times and can be conditional.

In the do-while-loop, t_c computes the result for $F_0(\sim)$ and stores it in the variable c. This result is written into s_c and is used in the condition of the do-while-loop of t_c, such that the do-while-loops of $t_{0'}$ and t_c will perform the same number of iterations. Task $t_{0'}$ reads the result from s_c, just before it needs to evaluate its condition.

In Figure 6.10, no assignment-statement is depicted in t_0, instead $\sim$ is depicted at line 3. Note that if $\sim$ in t_0 would be an assignment-statement, it would have a non-manifest access pattern. The access pattern should be non-manifest, otherwise the do-while-loop would be behaving like a manifest for-loop. Furthermore, the template depicts a single task $t_{0'}$ that reads results for the condition from s_c, whereas multiple tasks may be extracted from the body of a do-while-loop that all can read the results from s_c. If the

capacity of s_c is larger than 1, it may be possible to pipeline the execution of these tasks, such that they can be in different iterations of the do-while-loop.

6.4 Stream processing

In this section, we present the statements that need to be inserted into a task in order to perform stream processing. To perform stream processing, a task contains an endless loop that executes the encapsulated statements an infinite number of times. From the statements preceding an endless loop in the sequential NLP, we create one initialization task, because these statements only have to be executed once. From the body of an endless loop, multiple tasks can be extracted. For these tasks, we will present a template that defines the insertion of an additional counter variable that will be used by the communication and synchronization statements. This counter variable enables the tasks to use their windows independently from each other, such that the tasks may be executing different iterations of the endless loop. The presented template is an extension of the template presented in Section 6.1.

The outline of this section is as follows. First, a template that defines the construction of an initialization task is presented, in Section 6.4.1. Subsequently, Section 6.4.2 presents a template that defines the additional statements that need to be inserted for stream processing.

6.4.1 Initialization task

This section presents a template that defines the placement of synchronization and communication statements into the initialization task. The initialization task is executed preceding all other tasks in the system. It performs synchronization, for the windows in the tasks that have to be initialized one iteration ahead of the other windows. Furthermore, initialization statements from the sequential NLP are executed, to write initialization values into the CBs. Finally, a flag is set that indicates that the CBs have been initialized, such that the tasks can start executing.

The inter-iteration communication, discussed in Section 4.3.3, supports reading from and writing into locations of an array for the next iteration of the endless loop. Thus, at a certain point in time there are two versions of an array, one for the current and one for the next iteration of the endless loop. An array a_x will be replaced by a CB s_x. During one iteration of the endless loop a task will acquire and release all $\sigma(a_x)$ locations for the array in s_x. If the value at location k in a CB s_x should be read during the next iteration of the endless loop, we will write it at location $k + \sigma(a_x)$. A task that has to read this location for the current iteration of a_x has to slide its window $\sigma(s_x)$ locations through the CB and thus will get the location in its RW one iteration later. This implies that WWs and RWs for the next iteration for a CB s_x should not start at location zero in the CB, but at location $\sigma(a_x)$. In this way, they can access a logical location k, which is stored at the physical location $k + \sigma(a_x)$. Furthermore, the tasks that access the current iteration and start at location 0 in s_x cannot access this location, until they finished their first iteration.

```
1. ws = ~;
2. wo = ~;
3. while(1){
4.   ws@ = ~;
5.   ~ = rs@;
6.   wo@ = ~;
7.   ~ = ro@;
8. }
```

(a)

1. acquireS($\sigma(s_{ws})$, s_{ws},t_0); } Sliding WW in s_{ws}
2. releaseD($\sigma(s_{ws})$, s_{ws},t_0);
3. acquireD-nb($\sigma(s_{rs})$, s_{rs},t_1); } Sliding RW in s_{rs}
4. releaseS($\sigma(s_{rs})$, s_{rs},t_1);
5. acquireS($\sigma(s_{wo})$, s_{wo},t_3); } Overlapping WW in s_{wo}
6. releaseS($\sigma(s_{ro})$, s_{ro},t_4); } Overlapping RW in s_{ro}
7. **write**(s_{ws},m_{ws},∼);
8. **write**(s_{wo},m_{wo},∼);
9. releaseDL(m_{wo},s_{wo},t_3);
10. systemInitialized = 1;

(b)

Figure 6.11: (a) An NLP that performs stream processing and (b) a template that defines the placement of communication and synchronization statements into an initialization task to initialize the CBs s_{ws}, s_{rs}, s_{wo}, and s_{ro}

For the initialization of the windows of the tasks, we will use an initialization task. This task also assigns initial values to locations in the CBs. The initialization task signals all other tasks, after the system has been initialized, by setting a flag.

A possible NLP from which the initialization task in the template in Figure 6.11(b) could have been extracted is depicted in Figure 6.11(a). In this NLP, values for the next iteration are written into the arrays a_{ws} and a_{wo} and read from the arrays a_{rs} and a_{ro}. The arrays a_{ws} and a_{rs} are replaced by CBs with sliding windows and the arrays a_{wo} and a_{ro} by CBs with overlapping windows. Preceding the endless loop of the NLP, values are written into a_{ws} and a_{wo} and therefore the initialization task should contain these assignment-statements. The tasks t_0, t_1, t_2, and t_3 will be extracted from the assignment-statements in the body of the endless loop at the lines 4, 5, 6, and 7, respectively.

Figure 6.11(b) depicts a template that defines the placement of communication and synchronization statements into an initialization task. For this template we consider the CBs s_{ws} and s_{rs} with sliding windows and the CBs s_{wo} and s_{ro} with overlapping windows. The template contains **write** statements to write initial values into the CBs s_{ws} and s_{wo}.

The template for the initialization task first contains synchronization statements, to initialize the windows in the CBs of the system. In these CBs, WWs are initialized before RWs. Furthermore, for a CB s_x we set the pointers at $\sigma(s_x)$ instead of 0, by executing acquire and release calls for $\sigma(s_x)$ locations. Note that the acquire statement for a sliding RW uses an *acquireD-nb* statement, this is a non-blocking acquire statement for data in a CB. This statement is necessary for a CB with sliding windows, if a RW should be initialized for the next iteration, while the CB also contains a WW for the current iteration. If the blocking acquireD call would have been used, the RW cannot be moved past such a WW for the current iteration.

Succeeding the synchronization for the windows in the CBs of the system, the initialization task contains write statements, to write initial values into the CBs. These are

the statements that precede the endless loop in Figure 6.11(a). In these statements, the array accesses have been replaced by write statements for CBs. The write statement for s_{wo} with overlapping windows is immediately succeeded by a releaseDL statement, to set the full-bit for the written location, such that the location can be acquired for reading. Note that the write statements do not have to be encapsulated by acquire and release statements. This is not necessary, because no other tasks will be executed during the execution of these statements by the initialization task.

The last line of the initialization task sets the *systemInitialized* flag, to notify all tasks that the system has been initialized.

6.4.2 Inter-iteration communication

In this Section, we will extend the templates for inter-task communication with an offset counter, such that the tasks can process streams of values. For stream processing communication, the body of the task is encapsulated by an endless loop. Due to the offset counter used for a CB, arrays for different iterations of the endless loop can be stored and accessed in the CB. Thus, it is not necessary that only one copy of an array for the current iteration is stored and accessed in a CB.

A producer t_p that performs stream processing, will acquire all locations for an array a_x in its overlapping WW$_p$ in a CB s_x, for each iteration of its endless loop. Consider the case that we use a CB with $\theta(s_x)$ locations and perform an acquire call for each location in the array, i.e. $\sigma(a_x)$ acquire calls. After an iteration of the endless loop, the head of the WW$_p$ $\hat{w}_p$ will point to physical location $\sigma(a_x) \mod \theta(s_x)$ in the CB. The modulo operation is necessary in case $\hat{w}_p$ has been wrapped around after it reached the end of the buffer.

For the next iteration of the endless loop of a stream processing task, we can either reset the pointers and start acquiring from physical location 0, or we continue and acquire the location consecutive to $\hat{w}_p$. We will continue and acquire the location consecutive to $\hat{w}_p$, after an iteration, such that we can immediately start with the following iteration of the endless loop of t_p, without resetting the pointers for the windows. In this case, the location consecutive to $\hat{w}_p$ is labeled as logical location 0. In contrast, resetting all pointers in s_x to physical location 0 would have required t_p to wait until all other tasks that access s_x had reset their pointers. The tasks should have waited for each other, to avoid race-conditions due to locations that might get overwritten.

We define the *offset* o_x for an iteration k of a task t, as the location in CB s_x at which the value of location 0 in a_x will be written. For an iteration k of a task t, with an initial offset l in a CB s_x, the offset o_x is:

$$o_x = ((k \times \sigma(a_x)) + l) \mod \theta(s_x) \tag{6.6}$$

The initial offset l is the size of the communicated array a_x, if s_x is accessed for the next iteration.

Due to the offset counters, for CBs with overlapping or sliding windows, the producers and consumers are not forced to be in the same iteration of their endless loop.

Therefore, our CBs enable the execution of the tasks to be pipelined. A CB possibly contains multiple versions of an array, where each array is stored for a different iteration of the endless loop.

The template in Figure 6.12 is an extension of the template in Figure 6.1. The template in Figure 6.12 defines the placement of an offset counter o_x, for the CBs with overlapping windows. Furthermore, a loop is inserted that polls the *systemInitialized* flag. The presented template defines the placement of the offset counters and this loop for a CB with overlapping windows. These statements can be inserted in a similar way into the other templates for the insertion of communication via CBs that have been presented in this chapter. For such a template, the offset should be added to the index-expression of an accessed location in a CB, in a similar way as defined in Figure 6.12.

Preceding the endless loop, the template in Figure 6.12 contains a loop that polls the *systemInitialized* flag while it is zero. After *systemInitialized* has been set to 1 by the initialization task, this loop will break and the endless loop can start executing.

In Figure 6.12, the RW in CB s_r is read for the current iteration and therefore o_r is initialized to 0. A task t that uses a window to access the next iteration initializes its offset o_x to $o_x = \sigma(a_x) \mod \theta(s_x)$. Therefore, the array for the current iteration can be written from location 0 until location $\sigma(a_x) - 1$. Note that the equation to initialize the offset contains the function $\theta(s_x)$, which returns the capacity of the buffer. In Chapter 7, we will discuss the computation of the buffer capacity, given a CSDF model.

In the template for stream processing, we see that during the processing phase the offset o_w is added to the index-expressions m_w for the written CB s_w. Furthermore, the offset o_r is added to the index-expressions m_r for s_r. By adding the offset to the index-expression, location 0 for the array a_x is mapped to the first location acquired for the iteration of the endless loop in s_x.

In the final phase of the template in Figure 6.12, the offset counters for the next iterations for the accessed CBs are computed. Instead of counting the number of iterations and computing the offset for the next iteration as present in Equation 6.6, we add the array size to the current offset and store the result modulo the buffer capacity as the offset for the next iteration.

6.5 Access types

In an NLP, the programmer can specify the access pattern that will occur in an array, by using a key word that specifies the access pattern type. Such an access pattern type determines the synchronization that is required for the CB that replaces this array. Though we cannot always derive an access pattern, due to non-manifest statements, an access pattern type may enable us to use a window that is smaller than the communicated array. In this section, we will present a template that defines the placement of synchronization statements for the FIFO access pattern type.

Section 4.3.5 discussed four classes of access pattern types, from which we currently only support the FIFO access pattern type. If it is specified for an array a_x that there will be a FIFO access pattern, then all tasks that access this array will not perform multiplicity,

```
int o_r = 0;
int o_w = σ(a_w) mod θ(s_w);
while(systemInitialized == 0){ }

while(1){
 int p = 0;
 acquireS(σ(a_w),s_w,t_o);                          }Initial phase

 control{
  p++;                                              }
  acquireDL(m_r+o_r,s_r,t_o);                       }
  write(s_w, m_w+o_w,read(s_r, m_r+o_r));           }Processing phase
  releaseDL(m_w+o_w,s_w,t_o);                       }
 }

 releaseS(σ(a_r),s_r,t_o);                          }
 o_r = (o_r + σ(a_r)) mod θ(s_r);                   }Final phase
 o_w = (o_w + σ(a_w)) mod θ(s_w);                   }
}
```

Figure 6.12: A template that defines the placement of the offsets variables o_r and o_w into a task t_o, for stream processing communication via the CBs s_r and s_w that contain overlapping windows

```
 t
 1. control{
 2.   acquireS(1,s_w,t);
 3.   acquireD(1,s_r,t);
 4.   write(s_w,m_w,read(s_r,m_r));
 5.   releaseD(1,s_w,t);
 6.   releaseS(1,s_r,t);
 7.   k_w++; k_r++;
 8. }
 9. for(i=0; i<max(η(t),σ(a_w),σ(a_r)); i++){
10.   if (i < σ(a_w) − k_w){
11.     acquireS(1,s_w,t);releaseD(1,s_w,t);}
10.   if (i < σ(a_r) − k_r){
11.     acquireD(1,s_r,t);releaseS(1,s_r,t);}
12. }
```

Figure 6.13: A template that defines the placement of synchronization and communication statements into a task t to read from the CB s_r and write into s_w for which a FIFO access pattern has been specified, such that sliding windows can always be used

skipping, or out-of-order access. This determines that a task t_n that access a_x, will have a lead-in of zero ($d_1(t_n, s_x) = 0$) and a lead-out of zero ($d_2(t_n, s_x) = 0$). Therefore, no locations have to be acquired in s_x during the initial phase and the first access in s_x can be succeeded by releasing the written location. This implies that the release of a written location will not be delayed, such that it is always possible to use a CB with sliding windows, if a FIFO access pattern is specified.

The template to replace the array communication via a read array a_r and a written array a_w, for which the FIFO access pattern is specified, by a CB with sliding windows is depicted in Figure 6.13. In this template, s_r replaces a_r and s_w replaces a_w. The read access for a_r at location m_r is replaced by **read**(s_r,m_r). The write access into a_w of value v at location m_w is replaced by **write**(s_w,m_w,v). Note that it is the programmer his responsibility that the index-expressions are such that the locations are accessed in FIFO order. If the task performs stream processing, an offset will be added to this index-expression, as explained in Section 6.4.

In the template in Figure 6.13, the counters k_w and k_r are used to count the number of accesses in s_w and s_r, respectively. In the final phase, these counters are used to acquire and release the remaining locations in s_w and s_r, such that all pointers for the sliding windows have been increased with the size of the communicated array. Therefore, in the final phase, the number of iterations of the for-loop is determined by the maximum among the array size of the arrays for which the FIFO access pattern has been specified and the remaining locations in the other CBs ($\eta(t)$). In the for-loop, we determine for a CB for which the FIFO access pattern has been specified, if there are locations left to be

acquired and released. This is done, by comparing the difference of the array size and the access counter with the current iteration of the for-loop.

6.6 Conclusion

In this chapter, we presented templates that are suitable for the automatic insertion of inter-task communication and synchronization statements into the tasks of a dependency graph. By replacing arrays with CBs, a task graph is obtained that has the same topology as the dependency graph. The tasks of this task graph can be executed on a multiprocessor system. By using these templates, the placement of the synchronization statements is such that no race-conditions can occur.

We presented templates that define the placement of inter-task communication and synchronization statements, to use CBs with sliding windows. We also presented templates that define the placement of communication and synchronization statements for CBs with overlapping windows. For the first template that we presented, we inserted the inter-task communication and synchronization statements, such that non-manifest access patterns in the CB could be supported. Therefore, initially all locations from the communicated array are acquired in the window. In this window, the locations can be accessed according to a non-manifest access pattern. After the statements in the task that access the window, statements are inserted to release all location from this window. We presented a refinement of this template, to insert communication and synchronization statements into a task that will have a manifest access pattern in its CB. For a manifest access pattern, we can compute a window size that is typically smaller than the communicated array. The non-manifest loops and if-statements contain expressions for conditions for which locations in CBs have to be accessed. Templates have been presented that define the placement of communication and synchronization statements for these expressions. We also presented that for stream processing applications an additional initialization task may be required. Furthermore, for stream processing applications, the templates need to be extended to enable inter-iteration communication for the endless loop. Finally, a template that defines the placement of synchronization statements for a specified FIFO access pattern type has been presented. Such an access pattern type determines where synchronization statements can be placed in a task. Though we cannot always derive an access pattern, due to non-manifest statements, an access pattern type may enable us to use a window that is smaller than the communicated array.

CHAPTER 7

Temporal analysis model

Abstract - In this chapter, we will discuss our underlying temporal analysis model. We will identify in which cases the inter-task synchronization can be modeled in a CSDF model. In addition, we will present a modeling technique to model the synchronization that is represented by a hyperedge in the task graph by multiple edges in a CSDF model.

An underlying temporal analysis model for a task graph provides insight in the temporal behavior of the application. Therefore, such an analysis model can be used to compute sufficient resources for an application, to meet temporal constraints or to compute optimizations to reduce scheduling and synchronization overhead. In this chapter, we will present the derivation of a CSDF analysis model from the inter-task synchronization in a task graph. A counter intuitive aspect of our approach is that we model the synchronization rather than the communication. Therefore, the throughput is directly related to the synchronization and indirectly to the communication. With a CSDF model, we can compute sufficient buffer capacities to meet a given throughput requirement.

The computation of sufficient buffer capacities for a task graph to meet a given temporal requirement can typically not be done by computing buffer capacities per CB of the task graph. We will call the computation of buffer capacities per CB in isolation local analysis. We will examine why local analysis is not always a suitable option, by examining two issues: a performance issue and a deadlock issue. To address both issues, global analysis is required, such that all dependencies between the tasks are considered at once. To perform global analysis of the synchronization between the tasks in the task graph, we

require a temporal analysis model in which we can capture the synchronization behavior between all the tasks in the task graph.

A temporal analysis model that is suitable to model the synchronization behavior of stream processing applications is the cyclo static dataflow (CSDF) model [BELP96, WBS07]. A CSDF model contains actors that have cyclic changing firing rules. These cyclic changing firing rules can be used to model the synchronization calls of a task for the execution of one iteration of its endless loop.

We will derive a CSDF model, by identifying so called synchronization sections in the tasks. We choose a synchronization section, such that it corresponds with the execution of an assignment-statement and contains the synchronization calls that are performed for this assignment-statement. We will prove that we can model the synchronization calls performed in such a synchronization section with a single phase of an actor in a CSDF model.

We will present the derivation of an actor from a task and the usage of edges between these actors to model the inter-task synchronization. First, we will discuss the derivation of a CSDF model, given manifest synchronization behavior between the tasks of a task graph. We will discuss the modeling of the synchronization behavior between one producer and one consumer, via a CB with sliding windows. Next, we will present the modeling in case of overlapping windows. To model the synchronization between multiple producers and consumers via a CB, we will introduce a modeling technique. Such a technique is necessary, because a task graph can contain hyperedges, whereas a CSDF model only contains edges between two adjacent actors.

The derivation of a CSDF model from CBs accessed in non-manifest loops and if-statements, will be discussed briefly. For a CB with sliding windows that is used inside a non-manifest loop or if-statement, we can extract a CSDF model, because the synchronization calls are performed unconditionally. However, if a CB with overlapping windows has to be used in the body of a non-manifest loop or if-statement, we cannot capture the non-manifest synchronization behavior in the CSDF model.

For this chapter, the outline is as follows. We will first discuss shortcomings of local analysis, by examining two issues in Section 7.1. Next, Section 7.2 will introduce the CSDF model. In section 7.3, we will discuss the synchronization sections that are used to model the synchronization between tasks. The derivation of a CSDF model given manifest synchronization behavior between the tasks in a task graph is presented, in Section 7.4. Subsequently, we present the derivation of a CSDF model given non-manifest synchronization behavior, in Section 7.5. Finally, the conclusions are presented, in Section 7.6.

7.1 Shortcomings of local analysis

In this section, we will show that the computation of the buffer capacities requires global analysis of the synchronization in the task graph. The reason is that tasks in a task graph access multiple CBs and acquire locations in these CBs in a sequential order. As a result, there is a dependency between the locations acquired in the different CBs.

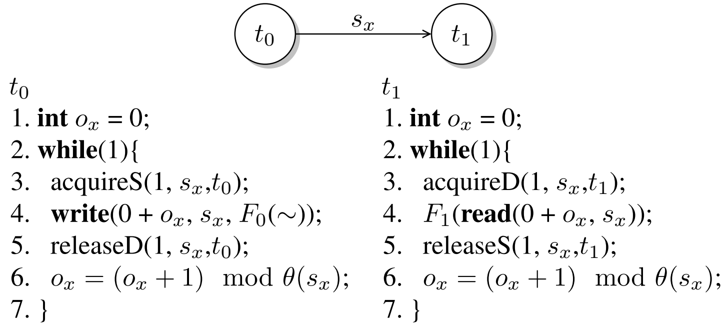


Figure 7.1: A task graph that contains a CB s_x with sliding windows, where the capacity of s_x determines the pipelined execution

For this section, the outline is as follows. In Section 7.1.1, we will demonstrate that buffer capacities computed per producer consumer pair can result in almost no pipelined execution of the tasks in a task graph. In Section 7.1.2, we will show that selecting a buffer capacity for a CB with overlapping windows that is smaller than the array size by using local analysis can result in deadlock.

7.1.1 Performance issue

In this section, we will show that considering one CB at a time for the computation of the buffer capacities possibly results in limited pipelined execution of the tasks in a task graph. Similar examples have been published by others.

Acquire statements have been inserted into the tasks of a task graph, to acquire locations in the accessed CBs, before the assignment-statement is executed. These acquire statements will block the execution of the assignment-statement, if they cannot return, due to the absence of the locations with data or space in their CBs.

Consider the task graph in Figure 7.1, with the tasks t_0 and t_1 that both perform stream processing and communicate via the CB s_x with sliding windows. If we set the capacity of s_x at one location ($\theta(s_x) = 1$), the tasks cannot be executed in parallel. Task t_0 can execute an iteration of its while-loop if it has the location in s_x acquired for writing and t_1 can execute an iteration if it has the location acquired for reading. There is no parallel execution, because the single location in s_x can be acquired by at most one task.

If we set the buffer capacity for s_x at two locations ($\theta(s_x) = 2$), the execution of the first iteration of t_0 can be succeeded by executing an iteration of both t_0 and t_1 in parallel. In this case, t_0 is one execution of its endless loop ahead of t_1, such that their executions are pipelined. Thus, by providing space to store two copies of the array a_x, the execution of the tasks can be pipelined.

Buffer capacities that enable the executions of the tasks of a task graph to be maximally pipelined cannot be selected per CB in isolation. Though, for the previous example

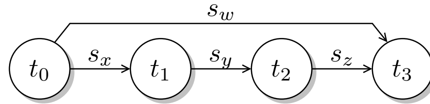


Figure 7.2: A task graph for which the execution of the tasks cannot be maximally pipelined, if the capacity of each CB is 2 locations

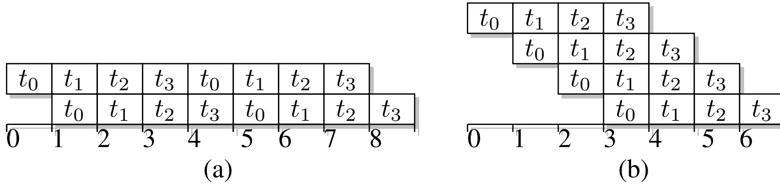


Figure 7.3: A schedule that illustrates the pipelined execution of the tasks from Figure 7.2, where (a) each CB can contain two arrays and (b) the CBs s_x, s_y, and s_z can contain two arrays and s_w can contain four arrays

we found that we could pipeline the execution of the two tasks by providing the tasks enough locations in the CB to store a copy of the array, this is in general not sufficient for a task graph. Even a buffer with a capacity larger than twice the size of the stored array may be required. To select buffer capacities that enable the executions of all tasks to be pipelined, global analysis of the task graph is required, as will be illustrated by the following example.

The task graph in Figure 7.2 depicts the tasks t_0, t_1, t_2, and t_3 that communicate via the CBs s_w, s_x, s_y, and s_z. Note that both t_0 and t_3 cannot execute, before they acquired a location in the two CBs that they access. Assume that each task acquires all locations for the communicated array at the beginning of an iteration of its endless loop and releases them at the end of an iteration. With this example, we will demonstrate that choosing the buffer capacity twice the size of the communicated array will not always lead to the maximum pipelined execution of the tasks.

If for each CB in Figure 7.2 we choose the capacity to be twice the size of the array that is communicated via the CB, at most two tasks can be executed in parallel. Figure 7.3(a) illustrates a schedule for the execution of these tasks. Because s_w can contain two arrays, t_0 can only be executed twice in a row and successively has to wait until t_3 released one array in s_w, before it can start another execution. Task t_3 cannot release space in s_w, before there is also an array released in s_z. Therefore, at least t_1, t_2, and t_3 have to be executed, before t_0 can be executed again. Figure 7.3(a) depicts that for these buffer capacities, at most two tasks can be executed in parallel.

By considering the blocking of t_0 due to the delayed release of data in s_w by t_3, we find that increasing the buffer capacity of s_w to the size of four arrays increases the amount of pipeline parallelism. Figure 7.3(b) depicts that for these buffer capacities all tasks can execute in parallel.

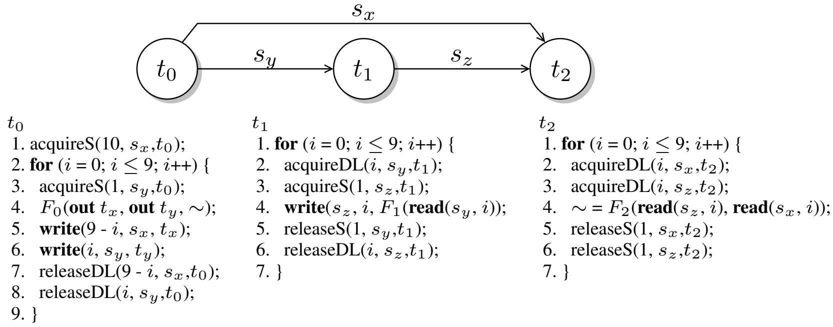


Figure 7.4: A task graph that contains CBs with overlapping windows, with an in-order access pattern without skipping or multiplicity for s_y and s_z

From this example, we can conclude that the computation of buffer capacities that enable all tasks to be executed in parallel, requires global analysis of the synchronization in the task graph.

7.1.2 Deadlock issue

In this section, we will demonstrate that a buffer capacity smaller than the array size should not be computed per CB, because this can result in deadlock for the task graph. This holds for a CB with sliding windows, but also for a CB with overlapping windows. We will illustrate this with a task graph that contains three CBs with overlapping windows, two via which values are communicated in FIFO order and one with an out-of-order access pattern.

Figure 7.4 depicts a task graph, with the tree tasks t_0, t_1, and t_2 that communicate via the CBs s_x, s_y, and s_z. Task t_0, t_1, and t_2 access the locations in s_y and s_z in-order, without skipping and multiplicity, such that there is FIFO communication via these CBs. For a CB in isolation, FIFO communication requires only one location in the buffer, because the producer writes the values in the same order as the consumer reads them, i.e. $\theta(s_y) = \theta(s_z) = 1$. The CB s_x is written out-of-order by t_0 and therefore requires a buffer capacity equal to the array size, i.e. $\theta(s_x) = \sigma(a_x) = 10$.

Because a task may perform acquire calls for different CBs before it releases locations in these CBs, there is a dependency between the acquire calls, such that the task graph in Figure 7.4 deadlocks if the CBs s_y and s_z both have a capacity of one location. The reason is as follows. Task t_0 starts by acquiring the whole array for its WW in s_x. Successively, it acquires a location for its WW in s_y and writes a value into s_x and s_y, after which it releases these locations. Task t_1 acquires the location in s_y in its RW and the location in s_z in its WW, then it reads from s_y and writes in s_z, after which it releases both locations. Subsequently, t_0 can acquire the location in s_y once more, after which it can write a value into s_y and s_x and release both written locations. Now both the location in s_y and in s_z contain a value and the full-bits for the locations 8 and 9 in s_x have been

set. But, t_2 will not acquire and release the location in s_z until it acquired location 0 in s_x. Location 0 in s_x will only be released by t_0, after it wrote and released the locations 7 until 1 in the CB s_x. But, to write and release a location in s_x, t_0 also has to acquire, write, and release the location in s_y. But, this location in s_y will not be acquired and released by t_1, before it can acquire and release the location in s_z, where the location in s_z has to be released by t_2. As a consequence, the three tasks are waiting for each other, such that deadlock occurs. To continue their execution, either the buffer capacity of s_y or s_z has to be increased. If additional values can be stored in one of these CBs, t_0 can write at location 0 in s_x, after which t_2 will start reading from s_z. In conclusion, the buffers are too small for deadlock-free execution, as a result of the dependencies between the acquire and release calls for the different CBs.

7.2 The CSDF analysis model

To perform temporal analysis, we will capture the synchronization between the tasks in a CSDF model. In this section, we will describe the CSDF model.

For the temporal analysis, we will use a dataflow analysis model [BELP96, LM87, LP95, PPL95, WBS07], because it is suitable for stream processing applications with cyclic dependencies. An often used dataflow analysis model is the synchronous dataflow (SDF) model [LM87]. In the SDF model, tasks can be described by actors. These actors communicate tokens via edges that represent queues, where a token models the synchronization between the tasks. In the SDF model, an actor has a single firing rule. An actor can be fired, if it can consume the number of tokens that is specified by its firing rule, from the queue of its incoming edges. Upon finishing its firing it will atomically produce tokens in the queues of its outgoing edges.

Figure 7.5(b) depicts a SDF model that corresponds with the cyclic task graph in Figure 7.5(a). In the SDF model, actor v_0 models t_0 and v_1 models t_1, the CBs s_w, s_x, s_y, and s_z are modeled by the edges e_w, e_x, e_y, and e_z. For this example, we assume that the CBs have an unbounded capacity. In the SDF model, for an incoming edge the number of consumed tokens is stated and for an outgoing edge the number of produced tokens.

The task graph and the SDF models in Figure 7.5 contain a cycle. In the task graph, such a cycle may be caused by t_1 that computes a condition for t_0. Task t_0 first sends a value to t_1 that successively computes the result of the condition given this value and sends the result back to t_0. In the SDF model, v_0 and v_1 have only a single firing rule, such that we have to model this synchronization by the production and consumption of one token. The consequence of the single firing rule is that the SDF graph deadlocks. Actor v_0 can only be fired if it can consume a token from e_w and e_y. After firing, v_0 will produce a token on e_x and e_z. Actor v_1 can only fire if it can consume a token from e_x. Because there are no initial tokens on the edges, the actors will not fire. Note that this cannot be solved by introducing initial tokens on these edges, because these tokens would not correspond to synchronization between the tasks.

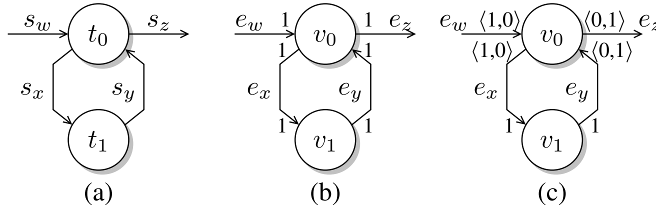


Figure 7.5: (a) A cyclic task graph modeled by (b) a SDF model that will deadlock and (c) a CSDF model that can model a cycle and does not deadlock

Instead of using the SDF model, will use the cyclo static dataflow (CSDF) model, which is a generalization of the SDF model [BELP96, PPL95, WBS07]. The CSDF model allows cyclic changing firing rules for each actor. We will use these cyclic changing firing rules, to model the different synchronization points in a task. With a CSDF model, the task graph in Figure 7.5(a) can be modeled. Figure 7.5(c) depicts a possible CSDF model, where some of the edges are annotated with a list of token consumption or production quanta. The lists with token consumption quanta represents the cyclic changing firing rules. In such a list, a position corresponds with a phase of the actor. For a firing of an actor, all token consumptions and productions for the corresponding phase are performed. In the CSDF model in Figure 7.5(c), v_0 will consume a token from e_w at the beginning of its first phase and at the end of this phase v_0 will produce a token on e_x. At the beginning of its second phase, v_0 consumes a token from e_y. The example shows that the CSDF model can be used to model the cycle in the task graph. This model is accurate enough to reflect the inter-task synchronization behavior, such that we will not draw the incorrect conclusion that the task graph deadlocks.

We use a CSDF model that consists of a directed graph $G = (V, E, \delta, \phi, \gamma, \pi, \lambda)$, with V the set of actors and E the set of directed edges. A directed edge $e_j = (v_h, v_i)$, with $e_j \in E$, is from actor v_h to actor v_i, with $v_h, v_i \in V$. An edge represents an unbounded queue. Actors communicate tokens over edges. There are $\delta(e_j)$ initial tokens on an edge e_j, with $\delta : E \to \mathbb{N}$. An actor v_i has $\phi(v_i)$ phases, with $\phi : V \to \mathbb{N}$, and transitions from phase to phase in a cyclic fashion. The first phase is phase 0 and the first firing is firing 0. An actor can be fired if all its input edges contain at least the number of tokens it will consume during that firing. The number of tokens consumed from an edge e_j by an actor v_i during firing l is determined by the current phase of the actor, so $\gamma(e_j, l \mod \phi(v_i))$ tokens are consumed, with $\gamma : E \times \mathbb{N} \to \mathbb{N}$. At the moment an actor v_i fires, it atomically consumes the tokens for the current phase from its input edges. The firing duration of v_i during phase n is the difference between the finish time and the start time of phase n and given by $\lambda(v_i, n)$, with $\lambda : V \times \mathbb{N} \to \mathbb{R}+$. Therefore, the firing duration of firing n of v_i is $\lambda(v_i, n \mod \phi(v_i))$. On finishing a firing, an actor atomically produces the tokens for the current phase on its output edges. The number of tokens produced during a phase g on e_j is $\pi(e_j, g)$, with $\pi : E \times \mathbb{N} \to \mathbb{N}$.

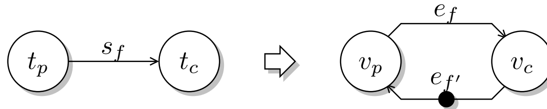


Figure 7.6: Communication via a FIFO buffer s_f and the extracted CSDF model

In [Wig09] the synchronization behavior for a task graph with the inter-task communication via FIFO buffers is captured in a CSDF model. The read and write calls for the FIFO buffers implicitly perform the acquire and release synchronization calls.

To be able to capture the synchronization behavior of a task graph conservatively in a CSDF model, the notion of a *non-blocking code segment* is introduced in [Wig09]. A non-blocking code segment is defined as a sequence of executed statements in the task that starts with a blocking acquire call and ends before the acquire call of the succeeding non-blocking code segment, or the end of the task. Thus, each acquire call starts a non-blocking code segment and the non-blocking code segment contains all statements executed preceding the execution of the next blocking acquire.

To extract a CSDF model from a task graph, each task is modeled by an actor. As illustrated in Figure 7.6, a FIFO buffer s_f between the task t_p and t_c is modeled by the two edges e_f and $e_{f'}$ between the actors v_p and v_c. The tokens produced on and consumed from e_f model data written into and read from the FIFO buffer s_f. The tokens on $e_{f'}$ model space in s_f and therefore this edge will contain initial tokens that represent the buffer capacity, as illustrated by the black dot.

The n-th non-blocking code segment of a task t_i that starts by acquiring l locations in s_f is modeled by the n-th phase of actor v_i that has a firing rule to consume l tokens from e_f, if s_f is read, or $e_{f'}$, if s_f is written. The locations released in FIFO buffer s_f during non-blocking code segment g are modeled by the production of tokens on e_f, if s_f is written, or $e_{f'}$, if s_f is read.

We assume that each actor in a CSDF model has a self edge that contains one initial token. For each phase of the actor, one token is consumed from and produced on the self edge. This self edge is included to prevent multiple overlapping firings of an actor. This self edge forces the firings to be sequentialized. In the depicted CSDF models, we will not draw this self edge, but leaf it implicit.

In [Wig09], it has been proven that the constructed CSDF model is temporally conservative compared to the task graph. By proving that a task will not release a location later compared to the token production of the corresponding actor, it is concluded that the extracted CSDF model is a temporally conservative representation.

7.3 Synchronization sections

To model the inter-task synchronization in a CSDF model, we will rely on the notion of synchronization sections. A synchronization section is a generalization of a non-blocking code segment. Synchronization sections will be used, to capture the execution

of assignment-statements. We will prove that the CSDF model extracted when using synchronization sections is conservative with respect to the synchronization behavior in the task graph.

For the extraction of a CSDF model, we will use synchronization sections, rather than non-blocking code segments. Synchronization sections are used for two reasons 1) a relation can be maintained between the execution of an assignment-statement and the synchronization section and 2) we can choose synchronization sections, such that they result in less phases than when using non-blocking code segments. Using non-blocking code segments could result in a large number of phases, because for a task that communicates via CBs with windows, the execution of an assignment-statement may be preceded by multiple acquire statements. To extract a CSDF model, using non-blocking code segments, a phase would be extracted per acquire statement. When a CSDF model is extracted while using synchronization sections, multiple acquires can be modeled by a single phase. By using synchronization sections, we can also maintain a relation between the execution of the assignment-statement and the phases of an actor in the CSDF model.

We define a *synchronization section* as follows. A synchronization section can start with the execution of an arbitrary statement in the code of a task. This synchronization section can contain an arbitrary number of consecutively executed statements. If this synchronization section contains a release call, it has to end with the execution of a statement that precedes the first acquire call that succeeds this release call. Note that a synchronization section does not have to contain acquire or release calls, but that it cannot contain an acquire call that succeeds a release call.

The definition of a synchronization section allows us to choose the synchronization sections, such that they correspond with the executions of the functions in a task. Furthermore, synchronization sections are a generalization of non-blocking code segments. According to the definition of a synchronization section, we can choose them, such that they satisfy the definition of non-blocking code segments.

Figure 7.7 gives an example of how we can choose synchronization sections in a task. This figure depicts a task in which the value of the counter p represents the number of the synchronization section during the execution of the task. In this example, we see that the value of p is increased at line 5. Therefore, the second synchronization section for this task starts at line 6 and runs until line 5, so after the execution of the body of the for-loop it also includes the execution of the for-loop statements at line 3 and 4. The acquire call during the initial phase, each execution of the assignment-statement, and the release call during the final phase are captured in a synchronization section. In Chapter 6, the dummy counter p has been included in most templates, to indicate how we choose the synchronization sections given these templates.

We will choose a synchronization section, such that if it includes an assignment-statement that accesses s_x, the synchronization section may include an acquire call for s_x that precedes the assignment-statement and a release call for s_x that succeeds it. In a task, an assignment-statement is preceded by an acquire statement and succeeded by a release statement for the accessed CBs, due to the usage of templates that define the placement of synchronization statements. Therefore, in a synchronization section, an acquire call for s_x will precede a release call for s_x.

```
1. int k_y=0, p=0;
2. acquireD(1,s_y,t_n);
3. for (i=0;i<5;i++){
4.   for (j=0;i<2;j++){
5.     p++;
6.     if(k_y < 9) acquireD(1,s_y,t_n);
7.     acquireD(1,s_x,t_n);
8.     F_0(read(s_x,2*i+j), read(s_y,2*i+1-j));
9.     if(k_y > 0) releaseS(1,s_y,t_n);
10.    releaseS(1,s_x,t_n);
11.    k_y++;
12. }}
13. p++;
14. releaseS(1,s_y,t_n);
```

Figure 7.7: A task for which the counter p holds the number of the synchronization sections during the execution of the task

Figure 7.8 illustrates the relation between synchronization sections and the acquire and release calls of a task. This figure also shows the relation between synchronization sections and non-blocking code segments. The execution trace in Figure 7.8 represents acquire calls for s_0 and s_1 with acq(s_0) and acq(s_1). These acquire calls precede the first function call F_0. In the execution trace, rel(s_2), rel(s_3), and rel(s_4) represent the release calls for s_2, s_3, and s_4. The assignment-statement that calls the function F_1 is executed before rel(s_4). After executing rel(s_4), the task ends. The trace consists of two *non-blocking code segments*, with the first one ($n(0)$) between the acquire for s_0 and s_1 and the second ($n(1)$) between the acquire for s_1 and the release for s_4. The first non-blocking code segment is modeled by consuming a token from e_0, as illustrated with cons(e_0), which conservatively models acq(s_0). The start of the second non-blocking code segment is modeled by cons(e_1), to model acq(s_1) conservatively. The second non-blocking code segment is succeeded by the three token productions prod(e_2), prod(e_3), and prod(e_4) that model the releases for s_2, s_3, and s_4 conservatively.

The execution trace in Figure 7.8 can also be modeled, using *synchronization sections* instead of non-blocking code segments. We choose the first synchronization section ($s(0)$), such that it includes the function F_0 that is preceded by cons(e_0) and cons(e_1) that conservatively model the two acquire calls that precede F_0. This synchronization section is succeeded by prod(e_2) and prod(e_3) that model the rel(s_2) and rel(s_3) calls conservatively. The second synchronization section ($s(1)$) contains F_1 and is succeeded by prod(e_4) to model the release call for s_4 conservatively.

We define a dataflow model to be *temporally conservative* with respect to the modeled task graph, iff each release call for a location in the task graph is performed earlier or at the same time as the production of the corresponding token that models this location in the dataflow model.

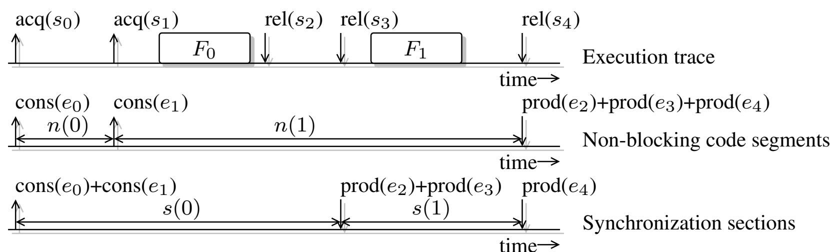


Figure 7.8: The relation between the acquire and release calls for a task and the production and consumption of tokens to model two non-blocking code segments ($n(0)$ and $n(1)$) and synchronization sections ($s(0)$ and $s(1)$)

In a dataflow model, an acquire call is modeled by consuming a token from an edge and a release call is modeled by producing a token on an edge. We will illustrate this with the SDF model in Figure 7.9(a) that models the acquire and release calls as they occur in the execution trace in Figure 7.8. Instead of using the CSDF model that contains an actor with multiple phases to model a task, for this example we will illustrate a single task with multiple SDF actors. Note that the actors in this SDF model can be folded into a single actor with multiple phases in the CSDF model.

Figure 7.9(a) illustrates a SDF model that directly models the acquire and release calls as they occur in the execution trace in Figure 7.8. Actor v_0 consumes tokens from its incoming edge, to model the acquire calls of locations in CB s_0. The function $e(i, e_0)$ returns the time at which the i-th token can be consumed from e_0, which models the time at which the i-th location can be acquired in s_0. Similarly, v_1 consumes a token from its incoming edge e_1 at time $e(i, e_1)$, to model the acquire call of the i-th location in s_1. The function $f(i, s)$ returns the time at which the i-th token is produced, which corresponds with the i-th location being released in s. Each actor v_i has a firing duration $\lambda(v_i)$, which corresponds with the worst-case execution time of the task between the events that are modeled by actor v_i.

The SDF model in Figure 7.9(b) is constructed from the synchronization sections in the task in Figure 7.8. The difference between the SDF models in Figure 7.9 is that in Figure 7.9(b) actor $v_{0'}$ is fired after it consumes a token from both e_0 and e_1 and has a firing duration of $\lambda(v_0) + \lambda(v_1) + \lambda(v_2)$, whereas in Figure 7.9(a) v_0 consumes a token from e_0 with firing duration $\lambda(v_0)$ and v_1 consumes a token from e_1 and has a firing duration $\lambda(v_1)$. Therefore, the firing of $v_{0'}$ is delayed until a token is available on both e_0 and e_1. Furthermore, the productions $\hat{f}(i, e_2)$ are $\hat{f}(i, e_3)$ are performed after the firing of $v_{0'}$ and are therefore delayed.

To prove that the extracted SDF model for synchronization sections is temporally conservative and will not result in deadlock, we will present three lemmas. The first lemma states that the delayed firing of an actor due to the token consumption from multiple edges will not result in earlier production times of tokens. The second lemma states that grouping actors with each an outgoing edge into a single actor will not lead to earlier

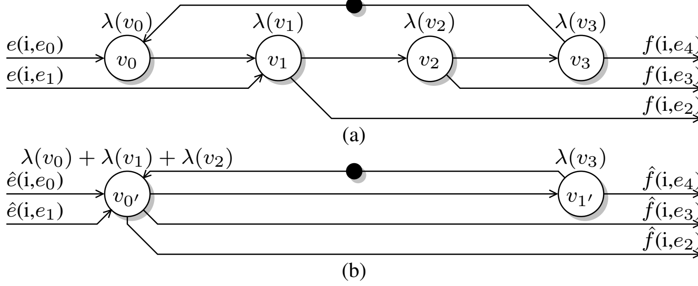


Figure 7.9: (a) The acquires and release calls in the execution trace of Figure 7.8 modeled in a SDF model and (b) the SDF model derived from the identified synchronization sections

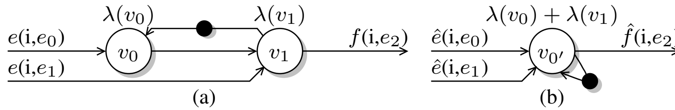


Figure 7.10: (a) A SDF model with the two actors v_0 and v_1 that both consume from an edge and (b) a SDF model derived from synchronization sections where these two actors have been merged into a single actor $v_{0'}$

token productions. The third lemma states that deriving a dataflow model from synchronization sections cannot introduce deadlock. Finally, a theorem combines these lemmas, to prove that the use of synchronization sections results in a conservative dataflow model.

Figure 7.10 is used to provide the intuition behind the first lemma. In this figure, the two actors v_0 and v_1 will be merged into $v_{0'}$, which has to consume tokens from both incoming edges before it can be fired. The first lemma states that modeling $v_{0'}$ is temporally conservative with respect to modeling v_0 and v_1 that are directly derived from the synchronization behavior of a task.

In Figure 7.10(a), the actors v_0 and v_1 model two consecutive blocking acquire operations of a task, by consuming tokens from e_0 and e_1. Actor v_0 can consume the i-th token from e_0 at $e(i, e_0)$, which models the moment that the i-th acquire call for the modeled CB can return successful. Similarly, v_1 can consume its i-th token from e_1 at $e(i, e_1)$. The actors v_0 and v_1 have a firing duration $\lambda(v_0)$ and $\lambda(v_1)$. Actor v_1 will produce its i-th token at $f(i, e_2)$, this models the i-th release call of the task for the modeled CB. In Figure 7.10(b), the actors v_0 and v_1 have been merged into $v_{0'}$, which has a firing duration that is the sum of the firing durations of the actors v_0 and v_1. For actor $v_{0'}$, the i-th token arrives at e_0 at time $\hat{e}(i, e_0)$ and at e_1 at time $\hat{e}(i, e_1)$. Actor $v_{0'}$ requires a token at both incoming edges before it can fire. It will produce its i-th token at e_2 at time $\hat{f}(i, e_2)$.

Figure 7.10 illustrates that $v_{0'}$ can only be fired if it can consume tokens from both input edges, whereas v_0 can be fired after it can consume a token from e_0, such that it

may be fired earlier than $v_{0'}$. Because the sum of the firing durations of v_0 and v_1 is equal to the firing duration of $v_{0'}$, this results in $v_{0'}$ not producing a token earlier than v_1.

Lemma 7.1. *The two actors v_0 and v_1 can be merged into a single actor $v_{0'}$, where the token production of $v_{0'}$ will not be earlier than the token production of v_1, because Equation 7.1 holds.*

$$e(i, e_0) \leq \hat{e}(i, e_0) \wedge e(i, e_1) \leq \hat{e}(i, e_1) \wedge f(i-1, e_2) \leq \hat{f}(i-1, e_2) \implies f(i, e_2) \leq \hat{f}(i, e_2) \quad (7.1)$$

Proof: From the SDF model in Figure 7.10(a), for the token production time of v_1 that is given by $f(i, e_2)$, we can derive the following equation:

$$f(i, e_2) = \max(e(i, e_1), \max(e(i, e_0), f(i-1, e_2)) + \lambda(v_0)) + \lambda(v_1) \quad (7.2)$$

The nested $\max$ expression can be removed from this equation, which results in the following equation:

$$f(i, e_2) \leq \max(e(i, e_1), e(i, e_0) + \lambda(v_0), f(i-1, e_2) + \lambda(v_0)) + \lambda(v_1) \quad (7.3)$$

By moving the constant $\lambda(v_0)$ outside the $\max$ expression, the following inequality is obtained for $f(i, e_2)$:

$$f(i, e_2) \leq \max(e(i, e_1), e(i, e_0), f(i-1, e_2)) + \lambda(v_0) + \lambda(v_1) \quad (7.4)$$

From the SDF model in Figure 7.10(b), for the token production time $\hat{f}(i, e_2)$ of actor $v_{0'}$, the following equation is derived:

$$\hat{f}(i, e_2) = \max(\hat{e}(i, e_1), \hat{e}(i, e_0), \hat{f}(i-1, e_2)) + \lambda(v_0) + \lambda(v_1) \quad (7.5)$$

It is given that $e(i, e_0) \leq \hat{e}(i, e_0)$, $e(i, e_1) \leq \hat{e}(i, e_1)$, and $f(i-1, e_2) \leq \hat{f}(i-1, e_2)$ and therefore $\max(e(i, e_0), e(i, e_1), f(i-1, e_2)) \leq \max(\hat{e}(i, e_0), \hat{e}(i, e_1), \hat{f}(i-1, e_2))$. From Equation 7.4 and Equation 7.5 it follows that $f(i, e_2) \leq \hat{f}(i, e_2)$. This results in the conclusion that Equation 7.1 holds. □

Lemma 7.1 is almost trivially generalized for the case that multiple actors, which each consume an input token from an edge, are merged into a single actor, which consumes these input tokens from these edges. Note that each actor that will be merged should also consume a token from an additional incoming edge and produce a token on an outgoing edge, such that these edges form a cycle and these actors are fired in a cyclic fashion. Figure 7.11 depicts that n actors that each consume an input token from one edge are merged into a single actor $v_{0'}$ that consumes input tokens from these n edges.

Figure 7.12 shows the intuition behind the second lemma. An actor v_0 and v_1 will be merged into an actor $v_{0'}$ that shows temporally conservative behavior.

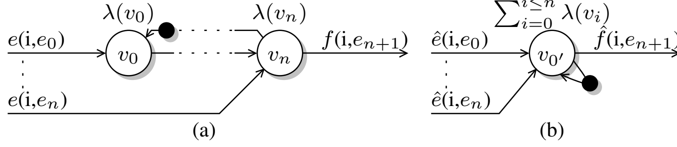


Figure 7.11: (a) A SDF model with the n actors that each consume from one edge from which the actors can be merged (b) into a single actor $v_{0'}$ that consumes from n edges

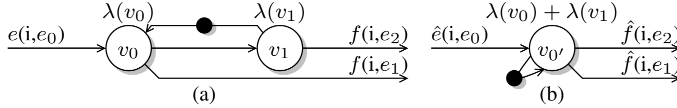


Figure 7.12: (a) A SDF model with the two actors v_0 and v_1 that both produce a token on an edge and (b) an SDF model as derived from a synchronization section where these two actors have been merged into a single actor $v_{0'}$

Lemma 7.2. *The two actors v_0 and v_1 can be merged into the actor $v_{0'}$, where the token productions of $v_{0'}$ will not be earlier than the token production of v_0 and v_1, because Equation 7.6 holds.*

$$e(i,e_0) \leq \hat{e}(i,e_0) \wedge f(i-1,e_1) \leq \hat{f}(i-1,e_1) \wedge f(i-1,e_2) \leq \hat{f}(i-1,e_2) \implies f(i,e_1) \leq \hat{f}(i,e_1) \wedge f(i,e_2) \leq \hat{f}(i,e_2) \quad (7.6)$$

Proof: From the SDF model in Figure 7.12(a), we derive the flowing equation for $f(i,e_1)$:

$$f(i,e_1) = \max(e(i,e_0), f(i-1,e_2)) + \lambda(v_0) \quad (7.7)$$

From the same dataflow model in Figure 7.12(a), we can also derive an equation for $f(i,e_2)$:

$$f(i,e_2) = \max(e(i,e_0), f(i-1,e_2)) + \lambda(v_0) + \lambda(v_1) \quad (7.8)$$

From the dataflow model in Figure 7.12(b), we can derive an equation for $\hat{f}(i,e_1)$ and $\hat{f}(i,e_2)$:

$$\hat{f}(i,e_1) = \hat{f}(i,e_2) = \max(\hat{e}(i,e_0), \hat{f}(i-1,e_1), \hat{f}(i-1,e_2)) + \lambda(v_0) + \lambda(v_1) \quad (7.9)$$

Because this equation states that $\hat{f}(i,e_1)$ and $\hat{f}(i,e_2)$ are equal, $\hat{f}(i-1,e_1)$ can be removed from this equation, which results in:

$$\hat{f}(i,e_1) = \hat{f}(i,e_2) = \max(\hat{e}(i,e_0), \hat{f}(i-1,e_2)) + \lambda(v_0) + \lambda(v_1) \quad (7.10)$$

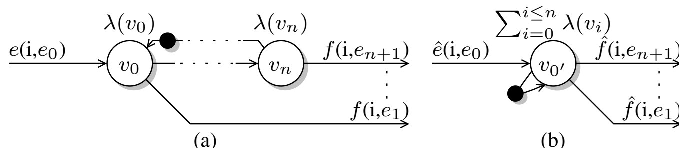


Figure 7.13: (a) A SDF model with n actors that each produce tokens at an output edge from which the actors can be merged (b) into a single actor $v_{0'}$ that produces tokens at the n output edges

We can conclude that $\max(e(i, e_0), f(i-1, e_2)) \leq \max(\hat{e}(i, e_0), \hat{f}(i-1, e_2))$, because it is given that $e(i, e_0) \leq \hat{e}(i, e_0)$, $f(i-1, e_1) \leq \hat{f}(i-1, e_1)$, and $f(i-1, e_2) \leq \hat{f}(i-1, e_2)$. Therefore, given Equation 7.7, Equation 7.8, and Equation 7.10, we can conclude that $f(i, e_1) \leq \hat{f}(i, e_1)$ and $f(i, e_2) \leq \hat{f}(i, e_2)$, such that Equation 7.6 holds. □

This lemma can also be trivially extended towards merging n actors. The intuition behind this extension is depicted in Figure 7.13. In this figure, n actors that each produce a token on their own output edge are merged into a single actor $v_{0'}$. This actor $v_{0'}$ produces a token on each output edge and has as firing duration that is the sum of the firing durations of the n merged actors.

Modeling a synchronization section can be seen as merging multiple actors as stated in Lemma 7.1 and Lemma 7.2. By merging these actors, we will not introduce deadlock in the dataflow model, because we only allow actors to be merged if the resulting actor corresponds to a synchronization section. In a synchronization section, acquire calls precede release calls. Therefore, merging actors cannot introduce new cycles, such that no deadlock will be introduced by modeling synchronization sections.

Lemma 7.3. *By modeling a synchronization section, which can contain several acquire and release calls, deadlock cannot be introduced in the dataflow model.*

Proof: Modeling synchronization sections will not introduce deadlock in the dataflow model, because no new cycles will be introduced. In a directed dataflow graph, merging an actor v_x and v_y, with an edge from v_x to v_y, can only introduce a new cycle if there is a second path from v_x to v_y. But this second path implies that v_x produces tokens on an outgoing edge, which corresponds to modeling a release call, and v_y consumes tokens from an incoming edge, which models an acquire call. The definition of a synchronization section states that it cannot contain an acquire call that succeeds a release call. If we model synchronization sections, we cannot merge the actors v_x and v_y, such that a new cycle cannot be introduced. Therefore, modeling synchronization sections cannot introduce deadlock. □

If we derive a CSDF model from the synchronization sections in a task, some acquire calls will be modeled by token consumptions that are delayed, due to the used synchronization sections. Furthermore, also some release calls will be modeled by delayed token productions. Lemmas 7.1 states that we can merge two actors, which corresponds with modeling tokens to be consumed later, while being temporally conservative. Lemma 7.2

states that we can merge two actors, which corresponds to later token productions, while being temporally conservative. Furthermore, Lemma 7.3 states that merging these actors to model synchronization sections will not introduce deadlock. Using these three lemmas, we can prove that by deriving a CSDF model from synchronization sections, the derived analysis model is temporally conservative towards the task graph.

Theorem 7.1. *The dataflow model extracted from a task by modeling its acquire and release calls in synchronization sections results in a temporally conservative dataflow model.*

Proof: In Lemma 7.1, it has been stated that modeling an acquire call as a token consumption that is performed later is temporally conservative, because a succeeding token production that corresponds with a release call can be performed no earlier than this release call. In Lemma 7.2, we state that modeling a release call as a token production that is delayed results in temporally conservative behavior, because none of the token productions will be performed earlier than their corresponding release call. Furthermore, Lemma 7.3 states that modeling synchronization sections will not introduce deadlock, because no new cycles are introduced. Therefore, we conclude that synchronization sections result in a temporally conservative dataflow model. □

7.4 Modeling manifest synchronization behavior

In this section, we will present the derivation of a CSDF model, given manifest synchronization behavior between the tasks in a task graph. We will first describe the derivation of a CSDF model for the manifest synchronization behavior between a single producer and consumer. The derivation of a CSDF model from the synchronization performed for a CB with sliding windows is described separately from the case where a CB with overlapping windows is used. Furthermore, because a CSDF graph is not a hypergraph, we will present a modeling technique to model the synchronization that is performed for the hyperedges in a task graph. We describe the derivation of a CSDF model for the synchronization via a hyperedge in three steps. The first step presents the extraction of a CSDF model given a CB with one WW and multiple RWs (OWMR). For the second step, we derive a model given a CB with multiple WWs and one RW (MWOR). For the third step, the two previous steps are combined to extract a CSDF model given the synchronization via a CB with multiple write and multiple read windows (MWMR).

The outline of this section is as follows. We will first describe the derivation of a CSDF model, given the synchronization preformed for a CB with one write and one read window (OWOR) for sliding and overlapping windows, in Section 7.4.1. Subsequently, Section 7.4.2 describes the extraction of a CSDF model, given the synchronization performed for a CB with OWMR. In Section 7.4.3, a CSDF model is derived given the synchronization performed for a CB with MWOR. Finally, we combine the two previous approaches to derive a CSDF model given the synchronization for a CB with MWMR, in Section 7.4.4.

7.4.1 One write window and one read window in a CB

In this section, we present the extraction of a CSDF model from the inter-task synchronization via a CB with OWOR. First, we explain the extraction of a CSDF model from the synchronization performed for a CB with sliding windows and subsequently from the synchronization performed for a CB with overlapping windows. The extraction of a CSDF model from the inter-task communication via a CB with sliding windows can also be found in [BBJS08] and for a CB with overlapping windows in [BBS09].

Sliding windows

In this section, we will describe the derivation of a CSDF model, given the synchronization performed for a CB with a sliding read and write window.

In a CSDF model, we model every task t_x by an actor v_x. A CB $s_h = (t_i, t_j)$ between a producer and a consumer is modeled by an edge pair, with an edge $e_h = (v_i, v_j)$ and a back-edge $e_{h'} = (v_j, v_i)$ between the actors v_i and v_j. Initially, $e_{h'}$ contains $\delta(e_{h'})$ tokens that represent empty locations and correspond to the capacity $\theta(s_h)$ of the modeled CB s_h. Edge e_h contains no initial tokens. The synchronization sections identified in a task t_x, as depicted by the counter p in the templates in the Figures 6.5 and 6.7, correspond to a phase of the corresponding actor v_x. Therefore, the number of phases ($\phi(v_x)$) of an actor v_x is equal to the total number of synchronization sections of the corresponding task. The firing duration $\lambda(v_x, i)$ of actor v_x during phase i corresponds to the worst-case execution time of the i-th synchronization section of t_x.

For a CB with *sliding windows*, Figure 7.14 shows an example of an extracted CSDF model from the inter-task synchronization between the tasks t_p and t_c. In Figure 7.14(b), the edges in the CSDF model are annotated with lists that represent the number of consumed or produced tokens during the different phases of the actors. A location in a list corresponds to a phase and the number corresponds to the number of consumed or produced tokens during that phase. The consumption of v_p from $e_{x'}$ is represented by the list $\langle 10 \times 1, 0, 0 \rangle$. This represents actor v_p consuming 1 token from $s_{x'}$ during phase zero until nine, because the 10×1 is a shorthand notation for 10 phases consuming 1 token per phase. Actor v_p consumes 0 tokens in the phases ten and eleven. Note that we simplify the depicted CSDF model by leaving the firing durations per phase of an actor implicit.

The consumption of k tokens by an actor from an edge models an acquireData call for k locations by a task and the consumption of k locations from a back-edge models an acquireSpace call for k locations. The production of k tokens by an actor on an edge models a releaseData call for k locations and the production of tokens on a back-edge models a releaseSpace call. The locations that are acquired during a synchronization section are modeled by tokens consumed at the start of a phase. The locations released during a synchronization section are modeled by the productions of tokens at the end of a phase. In Figure 7.14(b), the consumption of v_c from the edge e_x models locations that are acquired by acquireD calls in s_x by t_c. During each synchronization section, t_c acquires 1 location in s_x, which corresponds to the list $\langle 10 \times 1 \rangle$ in the CSDF model.

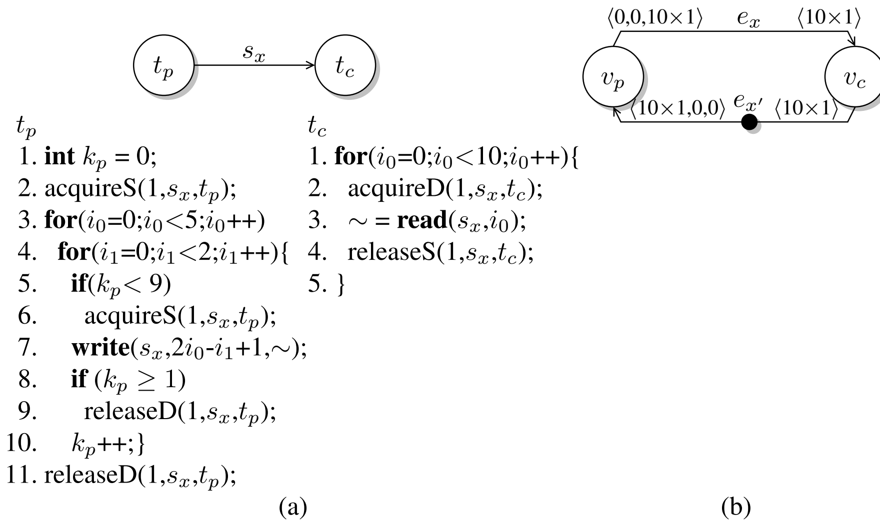


Figure 7.14: (a) A sliding read and write window in a CB from which the synchronization behavior is modeled in (b) a CSDF model

The tokens on an edge pair in a CSDF model correspond to the capacity of the modeled CB from a task graph. The tokens on the back-edge $e_{x'}$ represent locations without a value and the tokens on the forward edge e_x represent locations with a value that have not been read. In the CSDF model of Figure 7.14(b), during phase two on e_x a token is produced, this token corresponds to location zero that is released from the WW during synchronization section two by t_p. The token produced in the next phase of v_p, corresponds to the release of location one from the WW in s_x by t_p, etc.

With an extracted CSDF model, sufficient buffer capacities can be computed, by applying for example the algorithms proposed in [DIG99, SGB08, WBS07]. If we apply the algorithm in [WBS07] to the model in Figure 7.14 for a CB with sliding windows, we obtain that 3 initial tokens for $e_{x'}$ ($\delta(e_{x'}) = 3$) are sufficient. This corresponds to a buffer capacity of 3 locations ($\theta(s_x) = 3$) for the CB s_x, to guarantee deadlock-free execution of the task graph.

Overlapping windows

In this section, we will discuss the derivation of a CSDF model from the synchronization behavior between two tasks that communicate via a CB with overlapping windows. For a CB with overlapping windows, the derivation of a CSDF model from the synchronization behavior is less straightforward than for a CB with sliding windows. The reason is that an acquireData call or a releaseData call for a location will acquire or release a location

inside the window in the CB, instead of releasing the location at the tail of the WW or acquiring the location at the head of the RW.

For a CB with overlapping windows, the releaseS and acquireS calls are modeled in the same way as for a CB with sliding windows. The releaseS call for a consecutive location in s_x is modeled by the production of one token on the back-edge $e_{x'}$ in the CSDF model. The acquireS call for a consecutive location in s_x is modeled by the consumption of one token from the back-edge $e_{x'}$.

In a CSDF model, tokens are consumed in FIFO order from an edge. The tokens that are produced on a back-edge by an actor that models releaseS calls always represent consecutive locations. Therefore, the tokens consumed from this back-edge will represent consecutive locations. In contrast, a releaseDL or acquireDL call can acquire or release an arbitrary location between w and r. To model these calls, the order in which locations are acquired and released must be considered.

A *releaseDL* call for location k in s_x is modeled by the production of a token on e_x, where the produced token represents the released location k. The basic idea of modeling an *acquireDL* call for location k in the CB s_x, is to consume tokens from e_x until the token that represents the location k is consumed. For example, consider the releaseDL calls for the locations 0 and 1 in s_x that are modeled by producing a token representing location 0 and a token representing location 1 on an edge e_x. To model an acquireDL call for location 1, both tokens have to be consumed from e_x. If the next acquireDL call acquires location 0, this is modeled by consuming zero tokens, because the token representing location 0 has already been consumed in the previous phase.

Modeling an acquireDL call requires the lists with released and acquired locations. In Figure 7.15(a), the list $\{1, 0, 3, 2, 5, 4, 7, 6, 9, 8\}$ represents the locations released in s_x by t_p. The list of locations acquired by t_c is $\{0, 1, \ldots, 9\}$. In this task graph, task t_p does not release a location in synchronization section zero, it releases location 1 during synchronization section one, and it releases location 0 during synchronization section two. In the derived CSDF model, this is captured by v_p producing no tokens in phase zero, one token representing location 1 on e_x in phase one, and one token representing location 0 on e_x in phase two.

The acquireDL calls for locations in CB s_x by task t_c are captured by the consumption of tokens from e_x by v_c, in Figure 7.15(b). For the consumption from e_x, we use the shorthand notation $\langle 5 \times \langle 2, 0 \rangle \rangle$, which means that the pattern 2,0 is repeated five times, resulting in the list $\langle 2, 0, 2, 0, 2, 0, 2, 0, 2, 0 \rangle$. During phase zero of actor v_c, two tokens are consumed from e_x, to model the acquire of location 0, first the token that represents location 1 and next the token that represents location 0 is consumed. The acquireDL call in synchronization section one of t_c acquires location 1, actor v_c captures this by consuming zero tokens from e_x during phase one, because it already consumed the token representing location 1.

To model an acquireDL call of a consumer t_c from a CB s_x, we specify a function that returns the number of tokens to be consumed from an edge e_x. First we define the function $\hat{\beta}(t)$ that returns the number of synchronization sections that precede the processing phase of t_c, with $\hat{\beta} : T \to \mathbb{N}$.

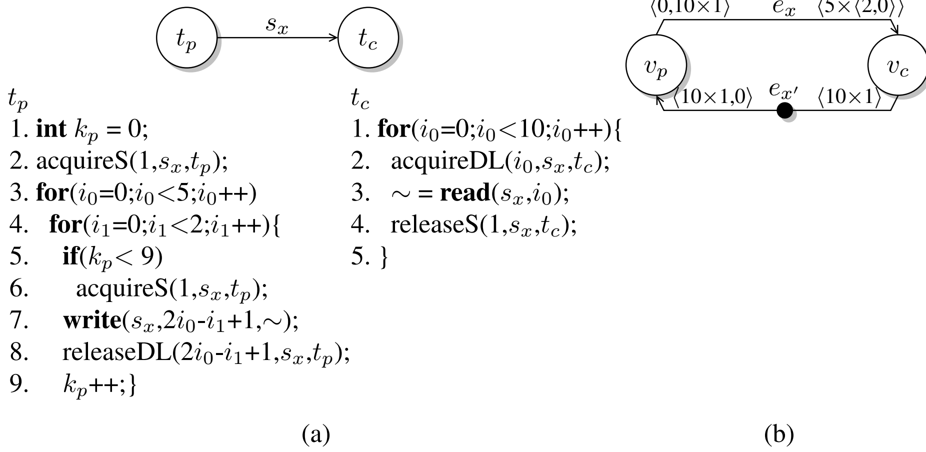


Figure 7.15: (a) An overlapping RW and WW in a CB and their synchronization behavior modeled in (b) a CSDF model

$$\hat{\beta}(t) = \begin{cases} \beta(t), \textbf{if } \max\{d_1(t,s) + d_2(t,s) | s = (Z_k, Z_l) \in S \wedge (t \in Z_k \oplus t \in Z_l)\} = 0 \\ \beta(t) + 1, \textbf{otherwise} \end{cases} \tag{7.11}$$

with the function $\beta(t_c)$ in Equation 6.2 returning the number of iterations for the for-loop in the initial phase of t.

We define the function $\omega(t_c, s_x, k)$ that returns the list with locations that are released by the producer t_p in s_x, before the location that will be read by the consumer t_c in synchronization section k is released, with $\omega : T \times S \times \mathbb{N} \rightarrow \{\mathbb{N}\}$.

$$\omega(t_c, s_x, k) = \{\alpha(t_p, s_x, l) \mid 0 \leq l \leq n; \alpha(t_p, s_x, n) = \alpha(t_c, s_x, k - \hat{\beta}(t_c))\} \tag{7.12}$$

For task t_c from Figure 7.15, $\omega(t_c, s_x, 0)$ results in {1,0}, because the producer writes locations 1 and 0 before the consumer can read location 0 during synchronization section zero. For synchronization section one, $\omega(t_c, s_x, 1)$ also results in {1,0}, because if the producer writes locations 1 and 0, the consumer can also read location 1.

We have to determine the locations that have to be acquired between synchronization section $k-1$ and k. This list is found by taking the relative complement (\) of the list with locations released preceding the location to be acquired in synchronization section k ($\omega(t_c, s_x, k)$) and the union of all locations released preceding the already acquired locations ($\bigcup_{l=0}^{l<k} \omega(t_c, s_x, l)$). By taking the cardinality (||) of the resulting set, i.e. the number of elements in this set, the function $\iota(t_c, s_x, k)$ returns the number of tokens to be consumed from e_x by actor v_c that models t_c during phase k, with $\iota : T \times S \times \mathbb{N} \rightarrow \mathbb{N}$.

$$\iota(t_c, s_x, k) = \mid \omega(t_c, s_x, k) \setminus \bigcup_{l=0}^{l<k} \omega(t_c, s_x, l) \mid \tag{7.13}$$

For task t_c from Figure 7.15, $\iota(t_c, s_x, 0)$ results in 2, because $\mid \{1,0\} \setminus \{\} \mid= 2$ and $\iota(t_c, s_x, 1)$ results in 0, because $\mid \{1,0\} \setminus \{1,0\} \mid=\mid \{\} \mid= 0$.

For an edge in the CSDF model, all produced tokens have to be consumed. Therefore, an actor that models acquireDL calls may need to consume tokens that represent skipped locations, whilst there are no corresponding acquireDL calls for these locations executed by task t_c. An actor v_c consumes the tokens for the skipped locations in the phase that corresponds to the first synchronization section of the final phase of the task.

With the extracted CSDF model from Figure 7.15, sufficient buffer capacities have been obtained, by applying the algorithm in [WBS07]. We obtained that 3 initial tokens for $e_{x'}$ ($\delta(e_{x'}) = 3$) are sufficient. These 3 initial tokens corresponds to a buffer capacity of 3 locations ($\theta(s_x) = 3$) for the CB s_x.

7.4.2 One write window and multiple read windows in a CB

This section presents an extension of the approaches presented in Section 7.4.1, to model a CB with OWMR. We show that we can model multiple consumers and still have a single back-edge to the actor that models the producer.

We allow multiple consumers in a CB. Therefore, a task graph can contain a hyperedge from one producer t_p to multiple consumers $\{t_{c0}, \ldots, t_{cr}\}$, as depicted in Figure 7.16(a). The OWMR CB, represented by this edge, cannot be modeled in a CSDF model with the previously mentioned approaches, because CSDF edges have one producing actor and one consuming actor. In the model that we derive from a CB with OWOR, the back-edge contains initial tokens that corresponded to the buffer capacity. If we model a CB with OWMR by making each consuming actor $\{v_{c0}, \ldots, v_{cr}\}$ have a back-edge to the producing actor v_p, there are multiple back-edges with initial tokens. For each of these back-edges, a different number of initial tokens can be computed, while a CB has only one capacity. In [DBC+07], a single back-edge to the producing actor v_p is derived, by merging the back-edges from $\{v_{c0}, \ldots, v_{cr}\}$ into a *merging actor* v_{mc}, from which there is a single back-edge to v_p.

Figure 7.16(b) shows the extracted CSDF model. The production and consumption quanta for the actors in this CSDF model are derived, by modeling each producer consumer pair for the hyperedge as a CB with OWOR, as presented in Section 7.4.1. Note that actor v_p uses the same list with produced tokens $\langle p_p \rangle$ for the edges to each of the consumers. From such an edge, an actor v_{cr} consumes with its own list with consumed tokens $\langle c_{cr} \rangle$.

The tokens produced on a back-edge by a consumer are consumed by a merging actor v_{mc}. This actor has one phase in which it produces and consumes one token and it has a firing duration of zero. Therefore, v_{mc} does not affect the behavior of the model. The actor v_{mc} only produces a token for the producing actor v_p, if each consuming actor

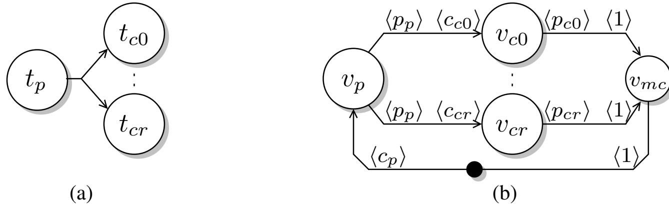


Figure 7.16: (a) A task graph with a CB with OWMR and (b) the derived CSDF model

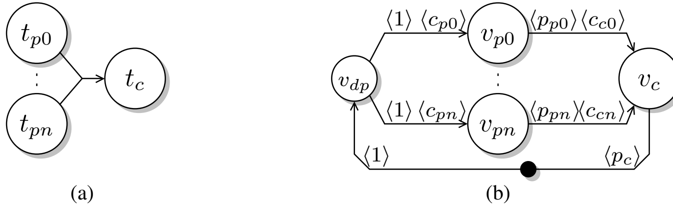


Figure 7.17: (a) A task graph with a CB with MWOR and (b) the derived CSDF model

$\{v_{c0}, \ldots, v_{cr}\}$ has produced a token. The production of a token by a consuming actor v_r, corresponds with a released location by a consumer t_r. Thus, the production of a token by v_{mc} models the case in which all the consumers, corresponding to $\{v_{c0}, \ldots, v_{cr}\}$, have released the location to which the token corresponds. The produced token becomes available to v_p, this corresponds to the location becoming available for task t_p to acquire.

7.4.3 Multiple write windows and one read window in a CB

In this section, we present the derivation of a CSDF model from a CB with MWOR. A counter-intuitive aspect is that in a CB with sliding windows a location l is written by at most one producer, whereas in the derived CSDF model all actors that model one of the producers for this CB will produce a token that corresponds to location l. Furthermore, to model a CB with overlapping windows, the lists with consumed tokens by an actor that models a consumer depends on the releases of all producers. The derivation of these CSDF models has a number of similarities with the derivation of a CSDF model from a CB with OWMR.

Figure 7.17 depicts the derivation of a CSDF model from the synchronization behavior between tasks via a CB with MWOR. The production and consumption quanta are derived for the actors in Figure 7.17(b), by modeling the producer consumer pairs for the hyperedge as CBs with OWOR. Figure 7.17(b) depicts that at the back-edge of the consumer, a distributing actors v_{dp} is inserted. If the actor that models the consumer produces a token, the distributing actor v_{dp} consumes this token and produces a token for each of the actors that model a producer. The token produced for an actor that models a producer corresponds to a location that can be acquired.

To model a CB with multiple overlapping write windows, the actor v_c has a different list $\langle c_{ci} \rangle$ per incoming edge. This is necessary, because for these edges each consumed token models a location that is acquired in the CB, instead of a consecutive location that is acquired for the non-overlapping RW. In the CSDF model in Figure 7.17(b), each token produced by one of the actors $\{v_{p0}, \ldots, v_{pn}\}$ represents a released location from an overlapping WW. For each phase, actor v_c consumes the token that corresponds to the location that is acquired during the modeled synchronization section for the overlapping RW. This implies that v_c may have to consume tokens among multiple edges, to model the acquire operations of locations released by different producers. Per phase, v_c will consume tokens from only a single edge, because only one actor will produce the token that models the location that is acquired. In contrast, to model sliding windows, an v_c has the same list with consumed tokens $\langle c_{ci} \rangle$ for each incoming edge, because each actor $\{v_{p0}, \ldots, v_{pn}\}$ will produce a token per modeled released location from its WW.

In [DBC+07], it is stated that they cannot model the behavior of multiple producers that communicate via one buffer in a CSDF model. Because we require the sequential code of an application to be in LSA form, a location in a CB is written at most once among the extracted tasks. This restriction results in the nice aspect that we can overlap the WWs of the producers, without causing race-conditions. As a result, we can model a CB with multiple write windows in a CSDF model.

Figure 7.18 depicts a task graph with an MWOR CB. Below this task graph, the extracted CSDF model is depicted. The producers are t_0 and t_1 and the consumer from s_x is t_2, these tasks are modeled by the actors v_0, v_1, and v_2, respectively. In the CSDF model, 3 tokens are sufficient for $e_{x'}$, i.e. $\delta(e_{x'}) = 3$. This corresponds to a buffer capacity of 3 locations for s_x, i.e. $\theta(s_x) = 3$.

7.4.4 Multiple write windows and read windows in a CB

In this section, we present the derivation of a CSDF model from a CB with MWMR, such that we can model the synchronization for a hyperedge in the task graph. Therefore, we combine the previous presented approaches to derive a CSDF model from a CB with OWMR and a CB with MWOR.

Figure 7.19(a) depicts a task graph with a CB with MWMR. In this task graph, $\{t_{p0}, \ldots, t_{pn}\}$ are the producers and $\{t_{c0}, \ldots, t_{cr}\}$ the consumers. We model each of the producers with an actor $\{v_{p0}, \ldots, v_{pn}\}$ and each of the consumers with an actor $\{v_{c0}, \ldots, v_{cr}\}$, as depicted in Figure 7.19(b). Similar as for the model derived in Figure 7.16, we add a merging actor v_{mc} to merge all tokens produced by the actors that model a consumer. We also insert a distributing actor v_{dp} that produces a token for each actor that models a producer, similarly as depicted in Figure 7.17. Between the actors v_{mc} and v_{dp} the derived model has a back-edge for which initial tokens can be computed.

For a CSDF model extracted from a CB with MWMR and sliding windows, an actor v_{cr} that models a consumer t_{cr} has the same list with token consumptions for each incoming edge. In that case, $\langle c_{r0} \rangle$ is equal to $\langle c_{rn} \rangle$. If the CSDF model has been extracted from a CB with MWMR and overlapping windows, during a phase k actor v_{cr} has to consume the token that models the location that task t_{cr} acquires during its k-th synchro-

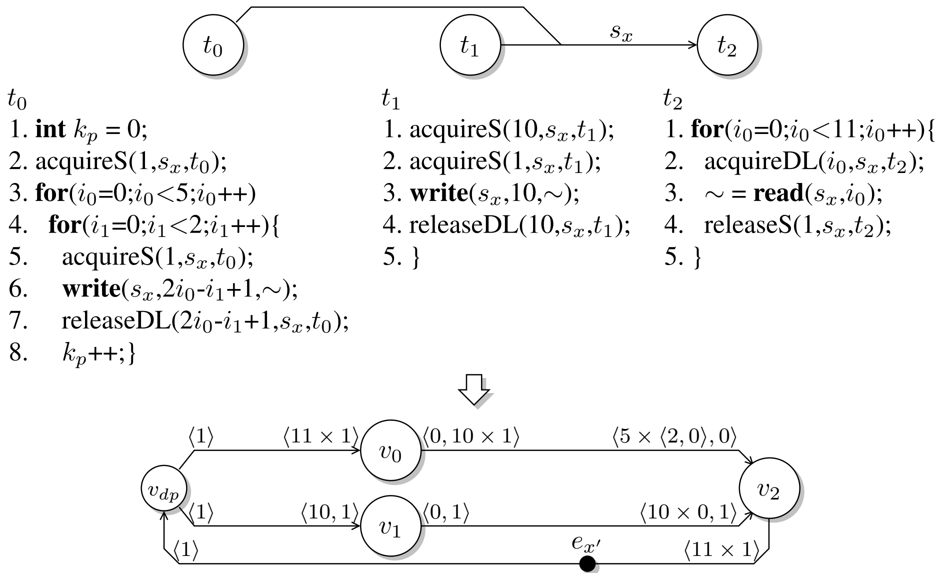


Figure 7.18: A CB with MWOR and overlapping windows, from which the synchronization behavior is modeled by a CSDF graph

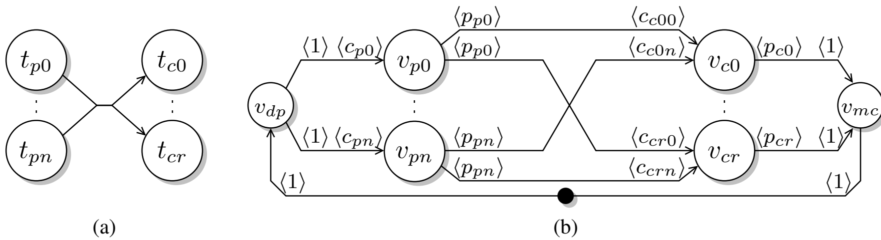


Figure 7.19: (a) A task graph with an MWMR CB and (b) the extracted CSDF model

nization section. This token will only be produced at one of the incoming edges, because for the modeled overlapping windows there will be at most one producer that will release this location. This situation is similar as explained in Section 7.4.3 and requires a different list with token consumption quanta per incoming edge for each actor that models a consumer.

7.5 Modeling non-manifest synchronization behavior

In this section, we present how non-manifest synchronization behavior can be modeled in a CSDF graph. Modeling non-manifest synchronization behavior is not straightforward, since the CSDF model only supports a static sequence of firing rules. Because the synchronization calls will be performed unconditionally for a CB with sliding window, the synchronization behavior is manifest. This manifest behavior can be captured in a CSDF model. For a task graph with a latency critical cycle, a CB with overlapping windows has to be used. If the synchronization calls for this CB are performed inside a non-manifest loop or if-statement, the synchronization behavior is non-manifest and cannot be captured in a CSDF model.

In the CSDF model, we can only express manifest synchronization behavior. A non-manifest loop or if-statement can result in a non-manifest number of synchronization sections. We model a synchronization section with a phase in the CSDF model. Because the CSDF model does not support a variable number of phases, we will use a single synchronization section for a non-manifest loop or if-statement. Note that loops and if-statements can be nested. In this case, we will model the outer-most non-manifest loop or if-statement as a single synchronization section.

For a non-manifest loop or if-statement with in their body an assignment-statement that uses a CB with sliding windows, the synchronization for the sliding windows will be performed unconditionally outside the non-manifest loop or if-statement. For such a non-manifest access pattern, the synchronization calls will be performed during the initial and final phase of the task. The executions of an assignment-statement inside the non-manifest loop or if-statement can be captured in a single synchronization section. We will model this synchronization section with a single phase in the CSDF model. As firing duration for this phase, we can use the worst-case execution time of all the executions of the assignment-statement that it models.

For a task graph that contains a cyclic data dependency that requires a CB with overlapping windows, where this CB is used inside the body of a non-manifest loop or if-statement, we cannot extract a CSDF model. Due to the cycle, we cannot replace the CB with overlapping windows by a CB with sliding windows, for which all synchronization calls would have been performed unconditionally. For the CB with overlapping windows, the acquireDL or releaseDL call will be performed in the body of a non-manifest loop or if-statement. To model the acquireDL or releaseDL calls in a CSDF model, we need their access patterns. But, these access patterns are non-manifest, due to the non-manifest loop or if-statement. These statements result in non-manifest synchronization behavior, which we cannot model in a CSDF model. Due to the latency critical cycle in the task

graph, modeling this synchronization behavior conservatively will probably result in a CSDF model that indicates deadlock, while the task graph is deadlock-free. Therefore, this is also not an option.

If the programmer specified the FIFO access pattern type for an array and this array is accessed in the body of a non-manifest loop or if-statement, then we cannot capture the synchronization behavior in a CSDF model. The reason is that for the FIFO access pattern type, the synchronization is inserted such that it immediately precedes and succeeds the assignment-statement. This results in non-manifest synchronization behavior that cannot be modeled in a CSDF graph.

7.6 Conclusion

In this section, we discussed short comings of local analysis and we discussed when the synchronization behavior between the tasks in a task graph can be modeled in a CSDF analysis model.

Local analysis does not consider all synchronization between the tasks in the task graph simultaneously and may therefore result in buffer capacities that are too small for deadlock-free execution or to meet a throughput constraint.

We model the synchronization behavior between the tasks in the task graph in a CSDF model, such that we can perform temporal analysis. The CSDF model can be used to compute sufficient buffer capacities for the CBs in the task graph. To model the synchronization behavior of a task, we identify synchronization sections in the code of the task. We define synchronization sections, such that they correspond with the execution of assignment-statements in a task.

Given the synchronization sections in a task, a CSDF model can be extracted. To extract a CSDF model from manifest synchronization behavior, we presented an approach to model the synchronization calls for CBs with sliding windows. Furthermore, we present an approach to model CBs with overlapping windows. We presented a modeling technique, to model a CB with multiple read and write windows in a CSDF model. Such a technique is necessary, because an edge in the CSDF model has two adjacent actors, whereas a CB in the task graph can have more than two tasks that access it. For a CB with sliding windows that is used inside a non-manifest loop or if-statement, we can extract a CSDF model, because the synchronization calls are performed unconditionally. If a CB with overlapping windows has to be used in the body of a non-manifest loop or if-statement, we cannot capture the non-manifest synchronization behavior in a CSDF model.

CHAPTER 8

Case studies

Abstract - In this chapter, we present an evaluation of our parallelization approach given two applications. We extract parallelism from a WLAN channel decoder and a JPEG decoder described in OIL. Both applications contain non-manifest loops and if-statements. For these applications, we extract a task graph for which we can compute buffer capacities, such that the execution of the tasks can be pipelined.

This chapter presents an evaluation of our parallelization approach, given two applications. These applications are a WLAN channel decoder and a JPEG decoder, both are described as an NLP in OIL. These applications contain non-manifest loops and non-manifest if-statements. An interesting observation is that both applications do only contain affine index-expressions. Furthermore, despite the observed cyclic data dependencies in both applications, CBs with sliding windows can be applied, because for these dependencies CBs with sliding windows will not introduce deadlock. In this chapter, we will first evaluate our approach for the WLAN channel decoder, followed by evaluating our approach for the JPEG decoder.

First, we will examine a WLAN channel decoder [IEE07] that is described as an NLP. Because the NLP that describes the complete WLAN decoder results in a large task graph, we demonstrate our approach using a simplified NLP of a WLAN channel decoder. For this simplified NLP, we will present the extracted task graph and the corresponding CSDF model. With the CSDF model, we will demonstrate that using different

temporal requirements results in different buffer capacities. We will illustrate that the buffer capacities are essential, in order to achieve task executions that overlap in time.

Our second case study is a JPEG decoder. This application is described as an NLP. The JPEG decoder contains a non-manifest loop, which results in a non-manifest cyclic data dependency. Therefore, we must apply CBs with overlapping windows. We cannot derive a CSDF model from the non-manifest synchronization behavior. As a programmer, we come to the conclusion that the arrays in the NLP are always accessed in FIFO order, such that we can specify the FIFO access pattern type. Therefore, we can use CBs with sliding windows. We also examined different buffer capacities and found that these influence whether the task executions can be overlapped. Besides that the JPEG decoder can be automatically parallelized, it can also be automatically mapped onto our 32 core MicroBlaze system.

The outline of this chapter is as follows. First, we evaluate our parallelization approach for a WLAN channel decoder, in Section 8.1. Subsequently, in Section 8.2, parallelism is extracted from a JPEG decoder. Finally, conclusions are presented, in Section 8.3.

8.1 WLAN channel decoder

A channel decoder application has been an important driver application for the work presented in this thesis. Therefore, we will evaluate our automatic parallelization approach, given a sequential description of a simplified WLAN channel decoder. For this channel decoder, we will present the extracted task graph and its corresponding CSDF model. With the CSDF model, we will compute for a number of temporal requirements the required buffer capacities. We will illustrate that sufficient buffer capacities are essential in order to overlap the executions of the tasks in time.

The sequential description of a channel decoder in OIL is depicted in Figure 8.1. This description in OIL contains an endless loop in the form of a *while(1)*, such that an endless stream of input values will be processed. Before the first iteration of the endless loop, the variable *state* has to be initialized. In the endless loop, the function *radioFrontend* obtains a value from the radio front-end. This function writes the value that it obtained from the radio front-end into the variable *sample*. Depending on the state, either the function *detect* or *decode* is executed for this sample. If the function *detect* is called, it determines if in the next iteration it continues detecting or that the function decode will be executed, by updating the variable *state* for the next iteration. If *decode* is executed, it reads a *sample* and updates *state* to indicate whether decoding should take place during the next iteration. Furthermore, if the function *decode* is executed, it writes the processed sample into the variable *value0*. The variable *value0* will be read by the function *processSample0* for the second processing step. The function *processSample0* writes its result into the variable *value1*, which is read by the function *processSample1* that sends its result to an output for this channel decoder.

The OIL description of the channel decoder contains a number of interesting aspects. First of all, in the switch statement, both case 0 and case 1 write into *state*, but the switch

```
 1. int sample;
 2. int state;
 3. int value0;
 4. int value1;
 5.
 6. state = 0;
 7. while(1) {
 8.   sample = radioFrontend();
 9.   switch(state) {
10.     case 0 : {
11.       state@ = detect(sample);
12.     }
13.     case 1 : {
14.       decode(sample, out state@, out value0 );
15.       value1 = processSample0(value0);
16.       processSample1(value1);
17. }}}
```

Figure 8.1: An OIL description of a channel decoder

statement guarantees that only one of the cases will be executed. This also means that the functions in a case of the switch-statement will not be executed for each iteration of the endless loop. Furthermore, for our approach, all tasks extracted from the switch-statement will read the variable *state* and at least the tasks containing the functions *detect* and *decode* have to write it. Therefore, buffers that support multiple reading and writing tasks are necessary.

Figure 8.2 depicts the task graph extracted from the OIL description in Figure 8.1. Task t_i preforms the initialization of the task graph, by writing the initial value into the CB s_{st}. This task contains the assignment-statement at line 6 in the OIL description. In this task graph, task t_f contains the function *radioFrontend* from line 8 in the OIL description. Task t_f gets an input value from the radio front-end and writes a sample into s_{sa}. Task t_{det} contains the function *detect* from line 11, reads samples from s_{sa}, and writes the state for the next iteration into CB s_{st}. Task t_{dec} contains the function *decode* from line 14. This task also reads samples from s_{sa} and writes the state for the next iteration into s_{st}. In addition, t_{dec} writes a decoded value into s_{v0}, where s_{v0} replaces *value0*. This CB s_{v0} is read by t_{p0} that executes the function *processSample0* from line 15. Task t_{v0} writes its result into s_{v1}. Task t_{p1} contains the function *processSample1* from line 16 and therefore reads from CB s_{v1} and sends its result to the output of the channel decoder.

To illustrate the internals of the tasks of the channel decoder, t_i and t_{dec} are depicted in Figure 8.3. Task t_i is depicted, because it is the initialization task. This task executes an acquire and a release call for the WWs of the tasks t_{dec} and t_{det}, such that they start writing in the next iteration for the variable *state*. Furthermore, the initial value 0 is

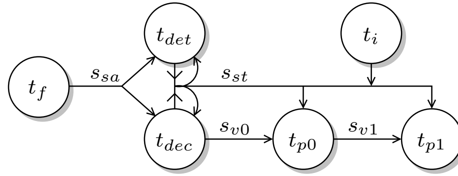


Figure 8.2: A task graph extracted from the channel decoder in Figure 8.1

written into s_{st}. After the initialization, the global integer *systemInitialized* is set to 1, such that the other tasks can start executing iterations of their endless loop.

In Figure 8.3, the task t_{dec} initializes its offset counters, where o_{stW} is initialized to $\sigma(s_{st}) = 1$, because this task writes the value for the next iteration of *state*. Note that we use the offset counters o_{stW} and o_{stR}, where o_{stW} is used for writing into s_{st} and o_{stR} for reading from this CB. At line 3, t_{dec} polls for *systemInitialized* to be set to 1, before it starts executing its endless loop. In the body of this endless-loop, at line 7 the condition for the non-manifest if-statement is computed and stored in the temporal variable *t*. This variable is used in the if-statement at line 13. The function call to *decode* at line 14, returns two values that are temporarily stored in t_{st} and t_{v0}. At line 15 and 16, these values are written into s_{st} and s_{v0}. At the end of the body of the endless loop, the remaining locations in the CBs are released and the offset counters for the CBs are increased.

The CSDF model extracted from the task graph in Figure 8.2 is depicted in Figure 8.4. In this model, the actors v_f, v_{dec}, v_{det}, v_{p0}, and v_{p1} model the task t_f, t_{dec}, t_{det}, t_{p0}, and t_{p1}, respectively. In this figure, the back-edges are labeled with $\delta(e_{sa})$, $\delta(e_{st})$, $\delta(e_{v0})$, and $\delta(e_{v1})$ that represent the number of initial tokens for these back-edges. The number of initial tokens that we will compute for such a back-edge corresponds to a sufficient buffer capacity for the CB the back-edge models. Note that for each modeled CB a merging and distributing actor have been added to the CSDF model.

The CSDF model of the channel decoder illustrates that the CSDF graph topology is not equal to the task graph topology. To model 5 tasks and 4 CBs, 13 actors and 29 edges are required. In the model, not only the back-edges contain initial tokens, also the outgoing edges of v_{det} and v_{dec} contain initial tokens. These edges model that the CB s_{st} is written for the next iteration. The CB s_{st} stores the value of the scalar variable *state* from the NLP. Note that in this CSDF model the self-edges are left implicit and the actors are not annotated with firing durations. This has been done to keep the figure concise.

With the extracted CSDF analysis model, buffer capacities can be computed, to meet a given temporal requirement in the form of a *throughput constraint*, using for example the approach presented in [WBS07]. For the CSDF models that we extract, 1 divided by the throughput constraint specifies the amount of time in which on average each actor v_i from the CSDF model should have fired for all its $\phi(v_i)$ phases. This corresponds to each modeled task having executed one iteration of its endless loop.

t_i

1. acquireS(1,s_{st},t_{dec});
2. releaseD(1,s_{st},t_{dec});
3. acquireS(1,s_{st},t_{det});
4. releaseD(1,s_{st},t_{det});
5.
6. **write**(s_{st}, 0, 0);
7. systemInitialized = 1;

t_{dec}

1. **int** t, t$_{st}$, t$_{v0}$;
2. **int** o_{sa} = 0, o_{stR} = 0, o_{stW} = $\sigma(s_{st})$, o_{v0} = 0;
3. **while**(systemInitialized != 1) { };
4.
5. **while**(1){
6.   acquireD(1,s_{st},t_{dec});
7.   t = **read**(s_{st},0 + o_{stR}) == 1;
8.   releaseS(1,s_{st},t_{dec});
9.
10.   acquireD(1,s_{sa},t_{dec});
11.   acquireS(1,s_{st},t_{dec});
12.   acquireS(1,s_{v0},t_{dec});
13.   **if** (t){
14.   decode(**read**(s_{sa}, 0 + o_{sa}), t$_{st}$, t$_{v0}$);
15.   **write**(s_{st}, 0 + o_{stW}, t$_{st}$);
16.   **write**(s_{v0}, 0 + o_{v0}, t$_{v0}$);
17.   }
18.   releaseS(1,s_{sa},t_{dec});
19.   releaseD(1,s_{st},t_{dec});
20.   releaseD(1,s_{v0},t_{dec});
21. o_{sa} = (o_{sa} + $\sigma(s_{sa})$) % $\theta(s_{sa})$;
22. o_{stR} = (o_{stR} + $\sigma(s_{st})$) % $\theta(s_{st})$;
23. o_{stW} = (o_{stW} + $\sigma(s_{st})$) % $\theta(s_{st})$;
24. o_{v0} = (o_{v0} + $\sigma(s_{v0})$) % $\theta(s_{v0})$;}

Figure 8.3: The initialization task t_i and task t_{dec} from the task graph in Figure 8.2

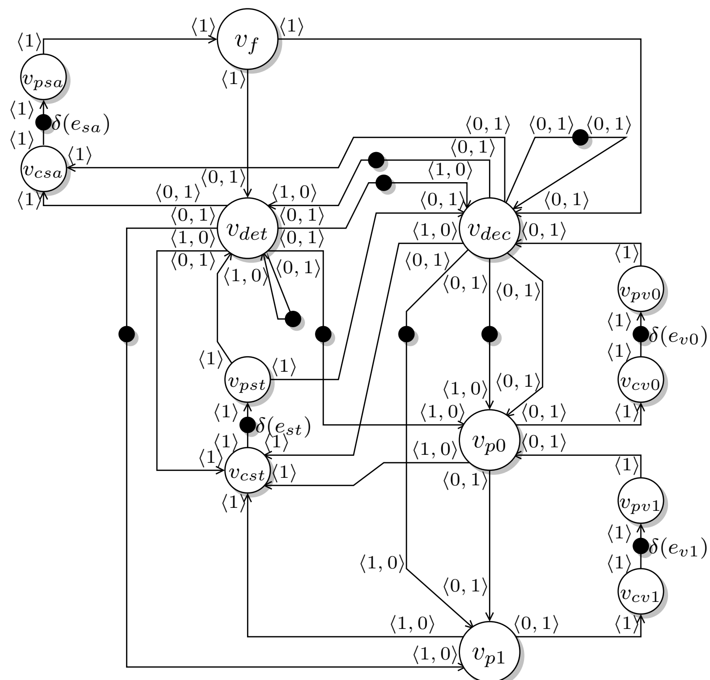


Figure 8.4: A CSDF model extracted from the task graph of the channel decoder depicted in Figure 8.2

To compute buffer capacities that are sufficient to meet a given throughput constraint, a firing duration should be assigned to each phase of an actor. This firing duration should correspond with the worst-case execution time of its synchronization section, in case that the tasks do not share processors. Our multiprocessor compiler cannot compute worst-case execution times, such that we currently assign a firing duration of one unit of time to each phase that corresponds to a synchronization section and 0 to the phase of each merging or distributing actor.

With the firing duration set to 1 or 0, we can compute a sufficient number of initial tokens, such that the actors in the CSDF model can be fired without deadlock. Additionally, the model enables us to examine the possibility to pipeline the executions of the tasks. If the desired throughput for the CSDF model is increased step-by-step, eventually the actor firings have to overlap in time. This requires additional initial tokens on the back-edges. The number of initial tokens that we compute corresponds to buffer capacities that enable the pipelined execution of the tasks of the application.

For four different throughput requirements, Table 8.1 depicts the computed buffer capacities for the buffers of the channel decoder in Figure 8.2. The table depicts that increasing the throughput from 4^{-1} to 2^{-1} does not require larger buffers. Note that for s_{st} two locations are required in the CB to store a value for both iterations. By increasing the throughput from 2^{-1} to 1^{-1}, the required buffer capacities double in size. This indicates that the pipelined execution of the tasks requires larger buffers.

Table 8.1: Computed buffer capacities ($\theta(s)$) to meet a throughput requirement in the form of executions per unit of time for the channel decoder in Figure 8.2

Throughput	s_{sa}	s_{st}	s_{v0}	s_{v1}
4^{-1}	1	2	1	1
3^{-1}	1	2	1	1
2^{-1}	1	2	1	1
1^{-1}	2	4	2	2

With the computed buffer capacities that are depicted in Table 8.1, we observed the schedule of the tasks with a dataflow simulator. This schedule is shown in Figure 8.5. For these executions, the firing duration for each synchronization section was set to 1 unit of time. We first executed the tasks with the buffer capacities computed to realize a throughput of 2^{-1} and subsequently executed the tasks using the buffer capacities computed to realize a throughput of 1^{-1}. The observed schedule for a throughput requirement of 1^{-1} clearly illustrates that, due to the larger buffers, the executions of the tasks can be pipelined. In contrast, for a throughput of 2^{-1} only 1 location is available in a buffer that is either acquired for reading or writing, such that there is less overlap in computations. The effect of the buffer capacities on the pipelined execution can clearly be seen for the number of executions of t_f after 25 units of time. Task t_f executed 12 times for a throughput requirement of 2^{-1} and 24 times for a throughput requirement of 1^{-1}.

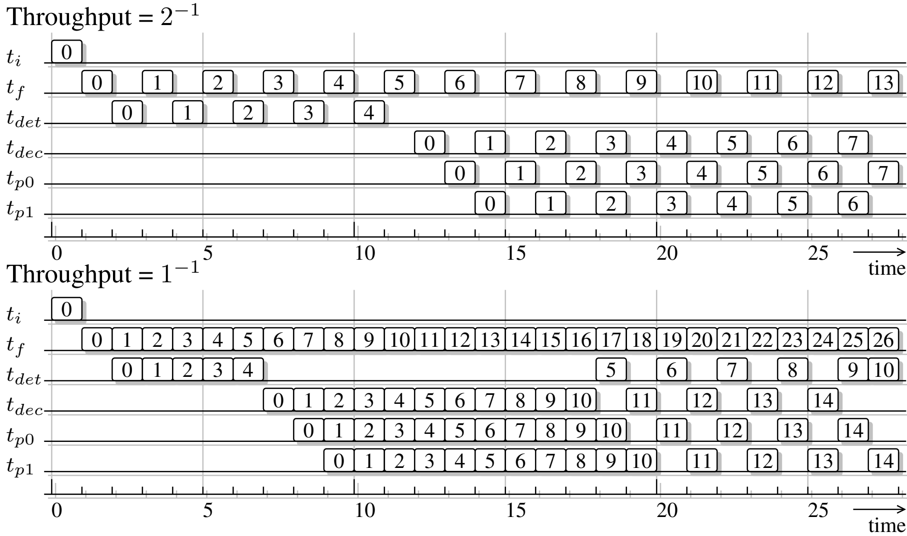


Figure 8.5: A schedule for a throughput requirement of 2^{-1} and 1^{-1} for the channel decoder application in Figure 8.2

For the extraction of the task graph, the extraction of the CSDF, and the computation of the buffer capacities for the discussed channel decoder, our parallelization tool Omphale required less than 2 seconds.

8.2 JPEG decoder

In this section, we evaluate our parallelization approach for a JPEG decoder. The OIL description of our JPEG decoder contains a non-manifest loop and a non-manifest if-statement. Seven tasks are extracted from the OIL description. Due to the non-manifest latency critical cycle, we have to apply CBs with overlapping windows. Therefore, we cannot extract a CSDF model. But as programmer, we observed that we can specify the FIFO access pattern type for the arrays in the OIL description. Due to the used access pattern type, CBs with sliding windows can be used and the required buffer capacities are significantly reduced. Though we cannot use the CSDF model to compute buffer capacities for different temporal requirements, we can examine the schedules for different buffer capacities. The obtained schedules clearly depict that we can pipeline the executions of the tasks.

The OIL description of our JPEG decoder is depicted in Figure 8.6. This description contains a non-manifest loop, such that images with up to 12289 frequency blocks can be decoded. In this code, first the *variableLengthDecodInitialize* (vldi) function is called that produces the dimension of the image in the variable d. Subsequently, the private variable i is set to 0. Note that no CB will be created for a private variable. In the

```
 1. Dimension d;
 2. FBlock fb[12289];
 3. PBlocks pbs[12288];
 4. PBlock pb[12288];
 5. private int i;
 6. d = variableLengthDecoderInitialize("input.jpg");
 7. i = 0;
 8. do{
 9.   fb[i] = variableLengthDecoder(d);
10.   if(last(fb[i])>0){
11.     pbs[i] = inverseDiscreteCosineTransformation(fb[i]);
12.     pb[i] = colorConversion(pbs[i],d);
13.     scaleImage(pb[i], d);
14.   }
15.   i = i + 1;
16. }while(last(fb[i-1])>0)
```

Figure 8.6: A JPEG decoder described in OIL

non-manifest loop, the *variableLengthDecoder* (vld) is called to write frequency blocks into array a_{fb}. Array a_{fb} is read by the *inverseDiscreteCosineTransformation* (idct) that outputs pixel blocks with YCbCr pixels in a_{pbs}. These pixel blocks are transformed by the *colorConversion* (cc) into one pixel block with RGB pixels that is written in a_{pd}. The function *scaleImage* (si) reads pixel blocks from a_{pb} and scales them to the resolution of the screen, for our multiprocessor system this is 800×640 pixels. The functions idct, cc, and si are encapsulated by a non-manifest if-statement that avoids that these functions will be called if $a_{fb}[i]$ contains the end of image (EOI) marker. The block is inspected using the function *last*. The same function is used in the condition of the loop, to verify that the last frequency block has been decoded. This function does not store state, such that we can use it at two places in the code.

By extracting parallelism from the OIL description in Figure 8.6, we get the task graph that is depicted in Figure 8.7. In this task graph, the task t_{vldi} contains the function vldi, t_{vld} the function vld, t_{idct} the function idct, t_{cc} the function cc, and t_{si} the function si. Task are extracted as well to evaluate the non-manifest conditions. Task t_{li} is extracted from the condition of the if-statement that calls the function *last* and the task t_{ll} is extracted from the condition of the loop that also calls the function *last*.

In the task graph, the arrays and variables have been replaced by CBs. The CB s_d replaces the variable *d*, s_{fb} replaces a_{fb}, s_{pbs} replaces a_{pbs}, and s_{pb} replaces a_{pb}. The CBs s_{ci} and s_{cl} have been added for the inter-task communication of the evaluated results for the conditions of the if-statement and the loop, respectively. Note that the tasks that contain functions that were in the body of the if-statement or loop in the OIL description, read the evaluated result for the condition from s_{ci} or s_{cl}, respectively.

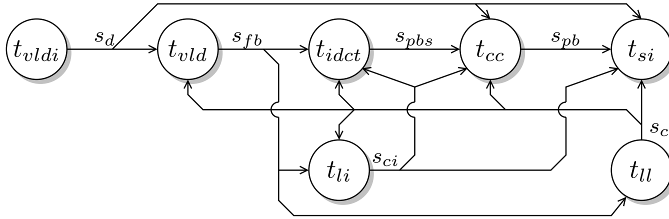


Figure 8.7: A task graph extracted from the JPEG decoder in Figure 8.6

The task graph of the JPEG decoder contains a latency critical cycle between t_{vld} and t_{ll}, via the CBs s_{fb} and s_{cl}. In the OIL description, this cycle is caused by the data dependencies between the assignment-statement at line 9 and the condition at line 16. The assignment-statement at line 9 writes into a_{fb}, which is read by the condition at line 16 and the result of this condition is necessary for the execution of the assignment-statement at line 9. For such a latency critical cycle, we can always apply CBs with overlapping windows, without introducing deadlock.

Due to the cyclic data dependency in the non-manifest loop, we must apply CBs with overlapping windows. In this case, the synchronization behavior will be non-manifest. This type of synchronization behavior cannot be modeled in a CSDF model. Therefore, we use CBs that have their array size as buffer capacity. The second row in Table 8.2 depicts the buffer capacities for the CBs with overlapping windows. Note that sliding windows will be used for s_{ci} and s_{cl}, because we automatically inserted them for the FIFO communication of the results of the conditions computed by t_{ll} and t_{li}.

Table 8.2: Buffer capacities ($\theta(s)$) for the CBs in Figure 8.7, using overlapping or sliding windows

	s_d	s_{fb}	s_{pbs}	s_{pb}	s_{ci}	s_{cl}
Overlapping windows	1	12289	12288	12288	1	1
Sliding windows	1	1	1	1	1	1

From the OIL description of the JPEG decoder, we extracted approximated dependencies for a_{fb}, a_{pb}, and a_{pbs}, because they are accessed in a non-manifest loop and if-statement. As a programmer, we can observe that in the OIL description the elements of a_{fb}, a_{pb}, and a_{pbs} are written and read in FIFO order. Therefore, we can specify the *FIFO access pattern type* for these arrays. Due to this access pattern type, these arrays will be replaced by CBs with sliding windows. Note that the synchronization calls for these windows will be performed immediately preceding and succeeding the execution of their assignment-statements, instead of conservatively calling these synchronization statements outside the loop or if-statement. This, makes it possible to use buffers capacities that are smaller than the array size.

```
t_cc
 1. int t, j, k_ci = 0, k_cl = 0;
 2. int i, k_pb = 0, k_pbs = 0;
 3. acquireD(1,s_d,t_cc);
 4. i = 0;
 5. do{
 6.   acquireD(1,sci,t_cc);
 7.   t = read(s_ci,k_ci++);
 8.   releaseS(1,s_ci,t_cc);
 9.   if (t){
10.     acquireD(1,s_pbs,t_cc);
11.     acquireS(1,s_pb,t_cc);
12.     write(s_pb, i, colorConversion(
 .       read(s_pbs,i),read(s_d,0)));
13.     releaseS(1,s_pbs,t_cc);k_pbs++;
14.     releaseD(1,s_pb,t_cc);k_pb++;
15.   }
16.   i = i + 1;
17.   acquireD(1,s_cl,t_cc);
18.   t = read(s_cl,k_cl);
19.   releaseS(1,s_cl,t_cc);
20. }while(t)
21. for(j = 0; j < 12288; j++){
22.   if (j < σ(s_pb) - k_pb){
23.     acquireS(1,s_pb,t_cc);
24.     releaseD(1,s_pb,t_cc);}
25.   if ( j < 1 )
26.     releaseS(1,s_d,t_cc);
27.   if ( j < σ(spbs) - k_pbs){
28.     acquireD(1,s_pbs,t_cc);
29.     releaseS(1,s_pbs,t_cc);}
30. }
```

```
t_li
 1. int k_ci = 0, k_fb = 0, k_cl = 0;
 2. int t, i, j;
 3. i = 0;
 4. do{
 5.   acquireD(1,s_fb,t_li);
 6.   acquireS(1,s_ci,t_li);
 7.   write(s_ci, k_ci++,
 .     last(read(s_fb,i)) > 0);
 8.   releaseS(1,s_fb,t_li);
 9.   k_fb++;
10.   releaseD(1,s_ci,t_li);
11.   i = i + 1;
12.   acquireD(1,s_cl,t_li);
13.   t = read(s_cl, k_cl++);
14.   releaseS(1,s_cl,t_li);
15. }while(t)
16. for(j = 0; j < 12289; j++){
17.   if ( j < (σ(s_fb) - k_fb)){
18.     acquireD(1,s_fb,t_li);
19.     releaseS(1,s_fb,t_li);
20.   }
21. }
```

Figure 8.8: Task t_{cc} and t_{li} from Figure 8.7 that use a FIFO access pattern type for s_{fb}, s_{pbs}, and s_{pb}

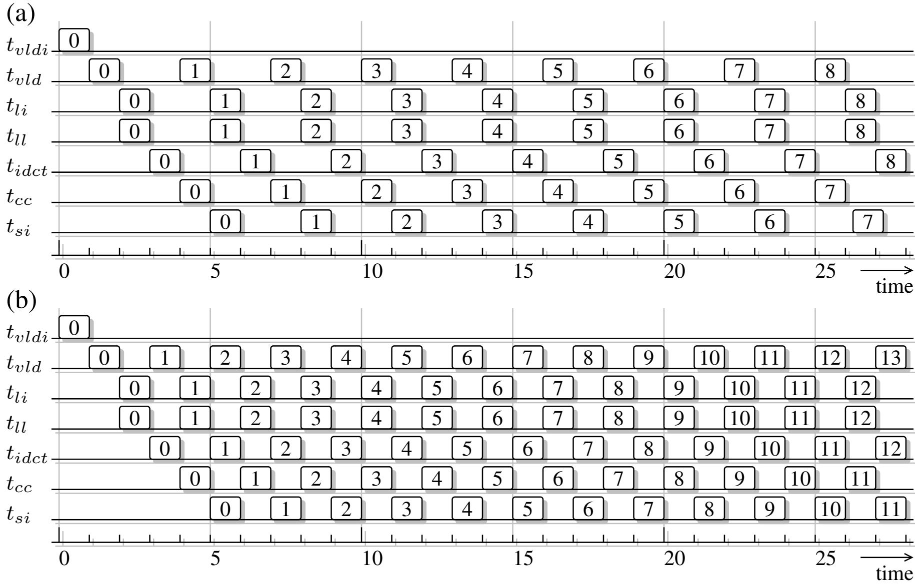


Figure 8.9: A schedule for the JPEG decoder, with (a) all buffer capacities set to 1 and (b) all buffer capacities set to the array size

Figure 8.8 illustrates the internals of tasks t_{cc} and t_{ci}. This figure shows the placement of the synchronization statements inside these tasks for the FIFO access pattern in s_{fb}, s_{pbs}, and s_{pb}. Both tasks contain the acquire and release statements for these CBs inside the non-manifest loop. Furthermore, each release statement for a CB s is succeeded by increasing an access counter k. During the final phase, this access counter will be used to release the remaining $\sigma(s) - k$ locations in the CB. Both tasks illustrate that the assignment to the private variable i is copied to each task that reads from i. In task t_{li}, the result for the condition of the non-manifest loop is required, to compute the result for the condition of the non-manifest if-statement.

From the JPEG decoder for which we specified the FIFO access pattern type, we cannot derive a CSDF model. Therefore, we cannot compute buffer capacities that are smaller than the array sizes or that guarantee a throughput constraint. But, because all locations are accessed in-order and written locations are released immediately, the application can execute deadlock-free if each buffer contains 1 location. These buffer capacities can be found in the third row in Table 8.2.

For the buffer capacities given in Table 8.2, we observed the schedules of the tasks, with a dataflow simulator. These schedules are shown in Figure 8.9. We used the task graph extracted from the OIL description that includes the FIFO access pattern types, such that the difference in monitored execution is purely a result from the different buffer capacities. Figure 8.9(a) shows the schedule of the tasks, given a buffer capacity of 1 location for each CB. Figure 8.9(b) shows the schedule of the tasks, given that the size of

the communicated array is used as the capacity for each CB. As for the channel decoder application, we do not use worst-case execution times for the tasks of the JPEG decoder. Instead, we set the firing duration of each synchronization section to 1 unit of time.

The schedules in Figure 8.9 clearly show that the overlap of executions for the tasks of the JPEG decoder is less if the buffer capacities are set to 1 location. In Figure 8.9(a), the executions for t_{vld} cannot overlap with the execution of t_{li}, t_{ll}, and t_{idct}. Their executions cannot overlap, because these tasks have to acquire the single location in s_{fb} for reading before they release it. In Figure 8.9(b), the executions of t_{vld} cannot overlap with the executions of t_{ll}, because they have a cyclic dependency via s_{fb} and s_{cl}. Note that due to this cyclic dependency, increasing the capacity of s_{cl} will not result in more overlap of task executions.

The extraction of parallelism from the OIL description of the JPEG decoder took our parallelization tool Omphale less than 6 seconds. Subsequently, our multiprocessor compiler assigned these tasks to processors and mapped the CBs into memories. Each task has been linked with a kernel and the communication libraries. After compilation, we obtained an executable for the JPEG decoder that executes correctly on our embedded multiprocessor system with 32 MicroBlaze cores.

8.3 Conclusion

In this chapter, we evaluated our parallelization approach for a WLAN channel decoder and a JPEG decoder. Both applications contain a non-manifest loop or a non-manifest if-statement. From these applications, parallelism could be extracted, such that the execution of their tasks could be pipelined. The pipelining of the task executions has been confirmed by schedules that we obtained by using a dataflow simulator.

From the channel decoder, a CSDF model could be extracted. With this temporal analysis model, we computed sufficient buffer capacities for a number of different throughput constraints. From the schedules for the different throughput constraints, we observed that for the highest throughput constraint the task executions are pipelined.

The JPEG decoder contains a latency critical cycle inside a non-manifest loop. Therefore, the extracted CSDF model is too conservative and deadlocks. For such a task graph, we can always use CBs with overlapping windows that have a buffer capacity equal to the size of the communicated array. But, after specifying the FIFO access pattern type, CBs with only 1 location were sufficient for deadlock-free execution of the task graph.

CHAPTER 9

Conclusion

This thesis presents an automatic parallelization approach to extract function parallelism from non-manifest stream processing applications, such that they can meet their real-time constraints, when executed on a multiprocessor system.

Stream processing applications are encountered in the channel decoding and video processing domain. These applications typically process an endless stream of input values and can contain non-manifest conditions and expressions. Furthermore, these applications typically have real-time requirements in the form of a throughput and a latency constraint. Stream processing applications are often executed on an embedded multiprocessor system.

We identified some important trends for both stream processing applications and embedded systems. For stream processing applications, the identified trends are that 1) they become more computational intensive and 2) they tend to contain an increasing number of non-manifest conditions and expressions. For embedded systems the identified trends are that 1) the number of processors on a chip is increasing and 2) multiple memories with different memory access latencies are used on a chip.

The identified trends have two clear implications, for the execution of stream processing applications on an embedded multiprocessor system. First of all the mapping effort increases, because the required processing power can only be satisfied by using multiple processors. Second, the validation effort increases, because the satisfaction of the temporal requirement has to be validated for an application that is executed on multiple processors.

To execute stream processing applications on a multiprocessor system, a multiprocessor compiler is required. This compiler maps a stream processing application onto an embedded multiprocessor system. If a multiprocessor compiler starts from a paral-

lel description of a stream processing application, the user has to manually partition the application, which can be time-consuming and error-prone. Instead, we focus on a multiprocessor compiler that starts from a stream processing application that is described in a sequential programming language. In this case, the multiprocessor compiler has to extract parallelism from the sequential description of an application.

In this thesis, we presented an automatic parallelization approach, to extract function parallelism from stream processing applications with non-manifest loops, if-statements, and index-expressions. We extract function parallelism, such that the execution of the tasks of an application can be pipelined, which will typically increase the throughput of the application. In the extracted task graph, the tasks perform inter-task communication and synchronization via circular buffers. Because the inter-task communication and synchronization statements are inserted automatically, no deadlock or race-conditions can be introduced by the programmer. Furthermore, these statements are inserted such that the inter-task synchronization can be captured in a temporal analysis model. With a temporal analysis model, system settings can be computed that are sufficient to guarantee the throughput constraint of a stream processing application. Examples of system settings that can be computed are the buffer capacities and scheduler settings.

The outline of this chapter is as follows. We will first present a summary of the results from the chapters of this thesis in Section 9.1, before we present an overview of the contributions of this thesis and future work, in Section 9.2 and 9.3, respectively.

9.1 Summary

In Chapter 1, we concluded that existing parallelization approaches are not suitable for the extraction of function parallelism from real-time stream processing applications with non-manifest statements. Existing approaches often cannot handle these applications, because they require manifest access patterns, such that they can apply first-in-first-out (FIFO) buffers for the inter-task communication, or they may require additional user input to indicate the correct insertion of the inter-task communication and synchronization statements. Furthermore, current parallelization approaches are often not extracting a temporal analysis model. As a consequence, temporal requirements are not taken into account during parallelization. These observations have motivated the development of a new parallelization approach that is presented in this thesis.

Related work is discussed in Chapter 2. In this chapter, we discussed alternative parallelization tools that start from a sequential programming language, extract parallelism, return a task graph described using a parallel programming language, and use an underlying temporal analysis model. We evaluated alternative sequential programming languages and concluded that the available sequential programming languages that do support non-manifest statements are not optimized for compile time analysis. For alternative parallelization approaches, we concluded that they typically cannot guarantee that a sufficient amount of function parallelism is extracted from an application. From the evaluation of several parallel programming languages, we concluded that for our purposes they do not provide a sufficiently rich application programming interface (API)

for inter-task communication and synchronization. Furthermore, we concluded that the cyclo static dataflow (CSDF) model seems to be an attractive temporal analysis model to be used in a multiprocessor compiler for stream processing applications. This model is attractive, because it supports cyclic dependencies.

In chapter 3, we presented the phases of our multiprocessor compiler that we implemented for the research that is presented in this thesis. The four phases of our multiprocessor compiler are the parallelization phase, the task analysis phase, the resource allocation phase, and the per processor compilation phase. Subsequently, we presented an overview of the parallelization phase. For the parallelization phase, we discussed three subphases: the dependency graph extraction phase, the inter-task communication and synchronization insertion phase, and the temporal analysis model extraction phase.

In Chapter 4, we concluded that for an application described in our Omphale input language (OIL) we can always derive the data dependencies between assignment-statements, at least at the array granularity. If we cannot derive at compile time for each execution of the reading and writing assignment-statements the array element that will be accessed, we extract an approximated data dependency. For an approximated data dependency it is assumed that each element of the array can be accessed. Furthermore, we concluded that for an application in single assignment form, it is sufficient to consider only the true data dependencies to determine a valid execution order of the assignment-statements. To enable the derivation of dependencies, the expressivity of OIL is restricted. This should not be an issue in practice, because OIL supports external functions that can be called, which can for example be described in C.

In Chapter 5, we introduced a new buffer type with overlapping windows, which has as feature that it can always be used to replace the communication via an array. This buffer supports multiple reading and writing tasks and thereby avoids the so called buffer selection problem. Because, given this buffer, a task can access array elements inside a window in an arbitrary order, the so called reordering problem is avoided. Furthermore, to reduce the synchronization overhead, we introduced a buffer with sliding windows, which can be seen as a special case of a buffer with overlapping windows. However, it is shown that this buffer is in some cases unsuitable for task graphs with cyclic data dependencies.

The templates that define where inter-task communication and synchronization statements must be inserted into the tasks have been presented, in Chapter 6. The placement of the synchronization statements is such that no race-conditions can occur. Furthermore, the synchronization statements are placed at positions in the application, such that we can model the synchronization behavior of the tasks in a dataflow analysis model.

In Chapter 7, we presented the extraction of a CSDF analysis model that captures the synchronization behavior of the tasks. We concluded that for a manifest application, we can always capture the synchronization behavior in a CSDF model. For a non-manifest application, we can model the synchronization behavior, if buffers with sliding windows have been used. However, if the application contains a non-manifest latency critical cycle, buffers with overlapping windows have to be used. In this case, the synchronization behavior will be non-manifest. This type of synchronization behavior cannot be modeled in a CSDF model. For the extraction of a dataflow analysis model, a number of modeling

techniques were presented. For example, a modeling technique is required, as a result of the different graph topology of a dataflow graph and a task graph. The reason is that a dataflow graph is not a hypergraph, whereas a task graph is a hypergraph. Therefore, we have to model one hyperedge in a task graph using multiple edges in a dataflow model.

In Chapter 8, we presented the evaluation of our parallelization approach, using a WLAN channel decoder and a JPEG decoder. We showed that function parallelism can be extracted from the WLAN channel decoder, such that the execution of the tasks could be pipelined, despite non-manifest statements. With the extracted dataflow analysis model, we were able to compute buffer capacities that maximized the pipelined execution of the tasks. We also extracted function parallelism from a JPEG decoder that contained non-manifest statements. Due to the latency critical cycle in the JPEG decoder, the CSDF model that we derived from this application was too conservative, such that the model deadlocked. However, by using our new buffer types for the inter-task communication, we showed that the execution of the tasks that we derived for the JPEG decoder could be pipelined.

9.2 Contributions

In this thesis we presented a new automatic parallelization approach to extract function parallelism from sequential descriptions of real-time stream processing applications. This approach relies on a new language to describe stream processing applications. This language allows non-manifest loops, if-statements, and index-expressions to be used. A key property of this language is that we can always derive the data dependencies at array granularity at compile time. Furthermore, an application described in this language always specifies a valid execution order, such that the extracted task graph will not deadlock. We introduced new buffer types that support multiple producers and consumers per buffer, such that we can always replace array communication by communication using these buffers. Because we can always derive the data dependencies and replace arrays by buffers, we can always extract the available function parallelism. Another new element is that our parallelization approach uses an underlying temporal analysis model in which we can capture the inter-task synchronization. This analysis model is used to compute system settings and to perform optimizations. The presented parallelization approach has been implemented in a multiprocessor compiler. The applicability of our parallelization approach has been evaluated using a WLAN channel decoder and a JPEG decoder.

9.3 Future work

Despite that the presented approach could be applied for a number of applications, we see many opportunities to generalize and improve it. Some of these possibilities are discussed in the following paragraphs.

In addition to the extraction of function parallelism, *data parallelism* can be extracted from some stream processing applications, where the underlying temporal analysis model

could potentially be used to indicate the points where parallelism can be extracted. Currently the underlying temporal analysis model is used to compute system settings to meet a given throughput constraint. Possibly, the temporal analysis model can be used, to indicate the tasks that are on a latency critical cycle in the application and the speedup that is required for these tasks to meet a given throughput constraint. Using this information, a parallelization approach could try to shorten this latency critical cycle, by extracting data parallelism.

Generalizations of our sequential programming language may make it easier to describe stream processing applications. Currently, OIL supports one endless loop, such that special techniques may be required to describe applications that may contain multiple loops that are potentially endless. A generalization of OIL to support multiple loops that are potentially endless has been proposed in [Geu10]. Our compiler would be applicable for a broader class of applications, if this generalization would be included.

Introducing *a more expressive dataflow model* than the cyclo static dataflow (CSDF) model may enable us to model the conditionally executed synchronization calls for overlapping windows. In a CSDF model, only manifest synchronization behavior can be modeled. Therefore, we cannot model the conditional synchronization for overlapping windows that are accessed in non-manifest loops or if-statements. The variable rate phased dataflow (VPDF) model [Wig09] is a more expressive dataflow model than the CSDF model. The VPDF model supports that the number of tokens that is consumed or produced during a phase of an actor is variable. However, a generalization of this model is needed to model the synchronization calls inside *nested* non-manifest loops.

By introducing an *underlying formal model for functional verification* of our sequential programming language OIL, we can verify and reason about the functional behavior of our applications. Our parallelization approach extracts parallelism, such that the functional behavior of an application described in OIL and the extracted task graph will be the same. Therefore, indirectly, such a model can be used to verify the functional behavior of a task graph. In contrast, verifying the functional behavior starting from a task graph can be difficult, because there can be a large number of execution orders of the tasks that might influence the functional behavior.

We can derive all data dependencies in OIL, such that it becomes possible to *allocate buffers to the local memories* in a multiprocessor system, instead of allocating all buffers to the central shared memory. Allocating buffers to local memories may significantly reduce the read access latency of the tasks that access these buffers, such that the execution time of these tasks is reduced. This can result in an increased throughput. Computing a suitable allocation of buffers to these local memories, such that the execution times of the tasks are minimized and the throughput is maximized is a potentially interesting optimization problem.

Bibliography

[AB09] S. V. Adve and H. J. Boehm. Memory model: A case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, 2009.

[ABC+95] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B.H. Lim, K. Mackenzie, and D. Yeung. The MIT alewife machine: Architecture and performance. In *Proc. Int'l Symposium on Computer Architecture (ISCA)*, pages 2–13, 1995.

[ACC+90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Int'l Conf. on Supercomputing (ICS)*, pages 1–6, 1990.

[ALSU06] A. V. Aho, M. S. Lam, R Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006. second edition.

[Ban94] U. Banerjee. *Loop parallelization*. Kluwer Academic publishers, Boston/ Dordrecht/ London, 1994.

[Bas03] C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *Proc. Int'l Symposium on Parallel and Distributed Computing (ISPDC)*, pages 23–30, Washington, DC, USA, 2003. IEEE Computer Society.

[BB07] J. W. van den Brand and M. J. G. Bekooij. Streaming consistency: a model for efficient MPSoC design. In *Proc. Euromicro Symposium on Digital System Design (DSD)*, pages 27–34, Washington, DC, USA, 2007. IEEE Computer Society.

[BBJS08] T. Bijlsma, M. J. G. Bekooij, P. G. Jansen, and G. J. M. Smit. Communication between nested loop programs via circular buffers in an embedded multiprocessor system. In *Proc. Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 33–42, New York, NY, USA, 2008. ACM Press.

[BBS09] T. Bijlsma, M. J. G. Bekooij, and G. J. M. Smit. Inter-task communication via overlapping read and write windows for deadlock-free execution of cyclic task graphs. In *Proc. Int'l Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, pages 140–148, Los Alamitos, CA, USA, July 2009. IEEE Press.

[BBS10] T. Bijlsma, M. J. G. Bekooij, and G. J. M. Smit. Circular buffers with multiple overlapping windows for cyclic task graphs. *Transactions on High-Performance Embedded Architectures and Compilers (HiPEAC)*, 2010. to appear.

[BCE+03] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. *Proc. of the IEEE*, 91(1):64–83, 2003.

[BCF97] D. Barthou, J. F. Collard, and P. Feautrier. Fuzzy array dataflow analysis. *Journal of Parallel and Distributed Computing*, 40(2):210–226, 1997.

[BELP96] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44[2]:397–408, 1996.

[Ber66] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, 1966.

[BMP+04] M. J. G. Bekooij, O. Moreira, P. Poplavko, B. Mesman, M. Pastrnak, and J. L. van Meerbergen. Predictable embedded multiprocessor system design. In *Proc. Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 77–91. Springer, 2004.

[BMvM07] M. J. G. Bekooij, A. J. M. Moonen, and J. L. van Meerbergen. Predictable and composable multiprocessor system design: A constructive approach. In *Bits & Chips Embedded System symposium*, October 2007.

[BPvM05] M. J. G. Bekooij, S. Parmar, and J. L. van Meerbergen. Performance guarantees by simulation of process networks. In *Proc. Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 10–19, New York, NY, USA, 2005. ACM.

[But02] G. C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Kluwer Academic publishers, Norwell, MA, USA, 2002.

[CCS+08] J. Ceng, J. Castrillon, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. MAPS: an integrated framework for mpsoc application parallelization. In *Proc. Design Automation Conference (DAC)*, pages 754–759, New York, NY, USA, 2008. ACM Press.

[CDVS07] J. Cockx, K. Denolf, B. Vanhoof, and R. Stahl. Sprint: A tool to generate concurrent transaction level models from sequential code. *EURASIP Journal on Advances in Signal Processing*, 2007(21):1–15, 2007.

[CFR+91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):490, 1991.

[CG06] M. Classen and M. Griebl. Automatic code generation for distributed memory architectures in the polytope model. In *Proc. Int'l Parallel and Distributed Processing Symposium (IPDPS)*, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[CGS99] D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann, San Francisco, 1999.

[CMM10] D. Cordes, P Marwedel, and A Mallik. Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming. In *Proc. Int'l Conf. on Hardware-Software Codesign and System Synthesis (CODES+ISSS)*, 2010. to appear.

[Col95] J. F. Collard. Automatic parallelization of while-loops using speculative execution. *Int'l Journal of Parallel Programming*, 23(2):191–219, 1995.

[DBC+07] K. Denolf, M. J. G. Bekooij, J. Cockx, D. Verkest, and H. Corporaal. Exploiting the expressiveness of cyclo-static dataflow to model multimedia implementations. *EURASIP Journal on Advances in Signal Processing*, 2007:1–14, 2007.

[DHRA06] M. Drake, H. Hoffmann, R. Rabbah, and S. Amarasinghe. MPEG-2 decoding in a stream programming language. In *Proc. Int'l Parallel and Distributed Processing Symposium (IPDPS)*, page 86, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[DIG99] A. Dasdan, S. S. Irani, and R. K. Gupta. Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems. In *Proc. Design Automation Conference (DAC)*, pages 37–42, New York, NY, USA, 1999. ACM Press.

[Dij65] E. W. Dijkstra. Cooperating sequential processes. Technical report, Department of Mathematics, Technological University of Eindhoven, 1965. Technical report EWD-123.

[dKSvdW$^+$00] E. A. de Kock, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. A. Vissers, and G. Essink. YAPI: application modeling for signal processing systems. In *Proc. Design Automation Conference (DAC)*, pages 402–405, New York, NY, USA, 2000. ACM.

[DM98] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[FCO90] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, 1990.

[Fea91] P. Feautrier. Dataflow analysis of array and scalar references. *Int'l Journal of Parallel Programming*, 20(1):23–53, 1991.

[Fea96] P. Feautrier. Automatic parallelization in the polytope model. *The Data Parallel Programming Model*, pages 79–103, 1996.

[FO05] B. Franke and M. F. P. O'Boyle. A complete compiler approach to auto-parallelizing C programs for multi-DSP systems. *IEEE Transactions on Parallel Distributed Systems*, 16(3):234–245, 2005.

[GBBC11] S. J. Geuns, M. J. G. Bekooij, T. Bijlsma, and H. Corporaal. Parallelization of while loops in nested loop programs for shared-memory multiprocessor systems. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2011. to appear.

[GC92] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, 1992.

[Geu10] S. J. Geuns. Parallelization of while-loops in nested loop programs for real-time multiprocessor systems, 2010. Master thesis, Eindhoven University of Technology, Eindhoven, The Netherlands.

[GNL01] O. P. Gangwal, A. Nieuwland, and P. E. R. Lippens. A scalable and flexible data synchronization scheme for embedded HW-SW shared-memory systems. In *Proc. Int'l Symposium on System Synthesis (ISSS)*, pages 1–6, New York, NY, USA, 2001. ACM.

[GSH88] K. Gharachorloo, V. Sarkar, and J. L. Hennessy. A simple and efficient implementation approach for single assignment languages. In *Proc. of conference on LISP and functional programming*, pages 259–268. ACM, 1988.

[HBC11] J. P. H. M. Hausmans, M. J. G. Bekooij, and H. Corporaal. Resynchronization of dataflow graphs. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2011. to appear.

[HGT07] K. Huang, D. Grünert, and L. Thiele. Windowed FIFOs for FPGA-based multiprocessor systems. In *Proc. Int'l Conf. on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 36–42, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[Hin01] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proc. Workshop on Program analysis for software tools and engineering (PASTE)*, pages 54–61, New York, NY, USA, 2001. ACM.

[HKH+09] W. Haid, M. Keller, K. Huang, I. Bacivarov, and L. Thiele. Generation and calibration of compositional performance analysis models for multi-processor systems. In *Proc. Int'l Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, pages 92–99, Piscataway, NJ, USA, 2009. IEEE Press.

[Hor97] S. Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):1–6, 1997.

[HRG+90] P. N. Hilfinger, J. Rabaey, D. Genin, C. Scheers, and H. De Man. DSP specification using the Silage language. In *Proc. Int'l Conf. Acoustics, Speech Signal Processing*, pages 1057–1060, 1990.

[HT07] K. Huang and L. Thiele. Performance analysis of multimedia applications using correlated streams. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 912–917, San Jose, CA, USA, 2007. EDA Consortium.

[IEE07] IEEE Computer Society. *IEEE 802.11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, June 2007. IEEE Std 802.11-2007.

[KC97] I. Karkowski and H. Corporaal. FP-Map - an approach to the functional pipelining of embedded programs. In *Proc. Int'l Conf. on High Performance Computing*, pages 18–21, Washington, DC, USA, 1997. IEEE Computer Society.

[KC02] M. Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *Proc. Design Automation Conference (DAC)*, pages 628–633, 2002.

[Kie00] B. Kienhuis. Matparser: An array dataflow analysis compiler. Technical report, University of California, 2000.

[KKS01] M. Kandemir, I. Kadayif, and U. Sezer. Exploiting scratch-pad memory using Presburger formulas. In *Proc. Int'l Symposium on System Synthesis (ISSS)*, pages 7–12, New York, NY, USA, 2001. ACM.

[KvMN+92] T. Krol, J. L. van Meerbergen, C. Niessen, W. Smits, and J. O. Huisken. The Sprite input language-an intermediate format for high level synthesis. In *Proc. Conf. on European design automation (EURO-DAC)*, pages 186 –192, 1992.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[Lan92] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[Lee06] E. A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.

[Lee07] E. A. Lee. Are new languages necessary for multicore?, 2007. Position Statement for Panel, Int'l Symposium on Code Generation and Optimization (CGO).

[Lei09] C. E. Leiserson. The Cilk++ concurrency platform. In *Proc. Design Automation Conference (DAC)*, pages 522–527, New York, NY, USA, 2009. ACM.

[LG95] C. Lengauer and M. Griebl. On the parallelization of loop nests containing while loops. In *Proc. Int'l Symposium on Parallel Algorithms/Architecture Synthesis (PAS)*, page 10, Washington, DC, USA, 1995. IEEE Computer Society.

[LM87] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

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

[LR92] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural aliasing. *Proc. conf. on Programming language design and implementation (PLDI)*, 27(7):235–248, 1992. SIGPLAN notices.

[LvMvdW+91] P. E. R. Lippens, J. L. van Meerbergen, A. van der Werf, W. F. J. Verhaegh, B. T. McSweeney, J. O. Huisken, and O. P. McArdle. Phideo: a silicon compiler for high speed algorithms. In *Proc. Conf. on European design automation (EURO-DAC)*, pages 436–441, Los Alamitos, CA, USA, 1991. IEEE Computer Society.

[Mes03] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface. http://www.mpi-forum.org/docs/docs.html, 2003.

[MHT+10] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. Comparison of parallelization frameworks for shared memory multi-core architectures. In *Proc. of the Embedded World Conference*, Nuremberg, Germany, 2010.

[MKTdK07] S. Meijer, B. Kienhuis, A. Turjan, and E. A. de Kock. A process splitting transformation for Kahn process networks. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1355–1360, San Jose, CA, USA, 2007. EDA Consortium.

[MNS10] S. Meijer, H. Nikolov, and T. P. Stefanov. Throughput modeling to evaluate process merging transformations in polyhedral process networks. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 747–752, San Jose, CA, USA, 2010. EDA Consortium.

[Moo09] A. J. M. Moonen. *Predictable Embedded Multiprocessor Architecture for Streaming Applications*. PhD thesis, Technical University of Eindhoven, June 2009.

[MP03] P. Magarshack and P. G. Paulin. System-on-chip beyond the nanometer wall. In *Proc. Design Automation Conference (DAC)*, pages 419–424, New York, NY, USA, 2003. ACM Press.

[NKG+02] A. Nieuwland, J. Kang, O.P. Gangwal, R. Sethuraman, N. Busá, K. Goossens, R. Peset Llopis, and P. E. R. Lippens. C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. In *Proc. Design Automation Conference (DAC)*, pages 233–270. Springer, 2002.

[NNS10] D. Nadezhkin, H. Nikolov, and T. P. Stefanov. Translating affine nested-loop programs with dynamic loop bounds into polyhedral process networks. In *Proc. Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 21–30, Piscataway, NJ, USA, 2010. IEEE Press.

[Ope08] OpenMP Architecture Review Board. OpenMP application program interface. http://openmp.org/wp/openmp-specifications/, 2008. version 3.0.

[ORS+06] G. Ottoni, R. Rangan, A. Stoler, M. J. Bridges, and D. I. August. From sequential programs to concurrent threads. *IEEE Computer Architecture Letters*, 5(1):6–9, 2006.

[PA98] G. A. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in Computers*, 46:329–400, 1998.

[PPL95] T. M. Parks, J. L. Pino, and E. A. Lee. A comparison of synchronous and cycle-static dataflow. In *Proc. of the Asilomar Conference on Signals, Systems, and Computers*, pages 204–210, Washington, DC, USA, 1995. IEEE Computer Society.

[Pth96] The posix threads standard, 1996. ISO/IEC standard 9945-1:1996, also known as ANSI/IEEE POSIX 1003.1-1995.

[Pug94] W. Pugh. Counting solutions to Presburger formulas: how and why. In *Proc. conf. on Programming language design and implementation (PLDI)*, pages 121–134, New York, NY, USA, 1994. ACM.

[PW94] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. In *Proc. Int'l Workshop on Languages and Compilers for Parallel Computing*, pages 546–566, London, UK, 1994. Springer.

[PW98] W. Pugh and D. Wonnacott. Constraint-based array dependence analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(3):635–678, 1998.

[Ram94] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1467–1471, 1994.

[RFGEL08] A. D. Reid, K. Flautner, E. Grimley-Evans, and Y. Lin. SoC-C: Efficient programming abstractions for heterogeneous multicore systems on chip. In *Proc. Int'l Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 99–108, New York, NY, USA, 2008. ACM.

[RP95] L. Rauchwerger and D. A. Padua. Parallelizing while loops for multiprocessor systems. In *Proc. Int'l Symposium on Parallel Processing (IPPS)*, pages 347–356, Washington, DC, USA, 1995. IEEE Computer Society.

[RvEP02] M. Rutten, J. van Eijndhoven, and E. J. Pol. Design of multi-tasking coprocessor control for Eclipse. In *Proc. Int'l Conf. on Hardware Software Codesign (CODES)*, pages 139–144, 2002.

[SBW09] M. Steine, M. J. G. Bekooij, and M. H. Wiggers. A priority-based budget scheduler with conservative dataflow model. In *Proc. Euromicro Symposium on Digital System Design (DSD)*, pages 37–44, Washington, DC, USA, 2009. IEEE Computer Society.

[SD03] T. P. Stefanov and E. F. Deprettere. Deriving process networks from weakly dynamic applications in system-level design. In *Proc. Int'l Conf. on Hardware-Software Codesign and System Synthesis (CODES+ISSS)*, pages 90–96, New York, NY, USA, 2003. ACM Press.

[SGB08] S. Stuijk, M. Geilen, and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, 57(10):1331–1345, 2008.

[SP09] R Selvaggi and L Pearlstein. Broadcom mediaDSP: A platform for building programmable multicore video processors. *IEEE Micro*, 29(2):30–45, 2009.

[Ste04] T. P. Stefanov. *Converting Weakly Dynamic Programs to Equivalent Process Network Specifications*. PhD thesis, Leiden University, The Netherlands, September 2004.

[TCA07] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *Proc. Int'l Symposium on Microarchitecture (MICRO)*, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[TCN00] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proc. Int'l Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101 –104, 2000.

[TE68] L. G. Tesler and H. J. Enea. A language design for concurrent processes. In *Proc. spring joint computer conference*, pages 403–408, New York, NY, USA, 1968. ACM.

[Tho95] S. Thompson. *Miranda: the craft of functional programming*. Addison-Wesley, 1995.

[TKD02] A. Turjan, B. Kienhuis, and E. F. Deprettere. Realizations of the extended linearization model in the Compaan tool chain. In *Proc. Int'l Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, pages 1–24, 2002.

[TKD04a] A. Turjan, B. Kienhuis, and E. F. Deprettere. An integer linear programming approach to classify the communication in process networks. In *Proc. Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 62–76, Berlin / Heidelberg, 2004. Springer.

[TKD04b] A. Turjan, B. Kienhuis, and E. F. Deprettere. Translating affine nested-loop programs to process networks. In *Proc. Int'l Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 220–229, New York, NY, USA, 2004. ACM Press.

[Tur07] A. Turjan. *Compiling Nested Loop Programs to Process Network*. PhD thesis, Leiden University, The Netherlands, March 2007.

[vdWdKH+04] P. van der Wolf, E. A. de Kock, T. Henriksson, W. M. Kruijtzer, and G. Essink. Design and programming of embedded multiprocessors: an interface-centric approach. In *Proc. Int'l Conf. on Hardware-Software Codesign and System Synthesis (CODES+ISSS)*, pages 206–217, New York, NY, USA, 2004. ACM.

[VJBC07] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, and F. Catthoor. A practical dynamic single assignment transformation. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12(4):40, 2007.

[VNS07] S. Verdoolaege, H. Nikolov, and T. P. Stefanov. PN: A tool for improved derivation of process networks. *EURASIP Journal on Advances in Signal Processing*, 2007(1):1–13, 2007.

[VSR96] I. Verbauwhede, C. Scheers, and J. M. Rabaey. Analysis of multidimensional DSP specifications. *IEEE Transactions on Signal Processing*, 44(12):3169–3174, 1996.

[Wal91] G. K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34(4):30–44, 1991.

[WBS07] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *Proc. Design Automation Conference (DAC)*, pages 658–663, New York, NY, USA, 2007. ACM Press.

[WEE+08] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem–overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

[Wig09] M. H. Wiggers. *Aperiodic Multiprocessor Scheduling for Real-Time Stream Processing Applications*. PhD thesis, University of Twente, Enschede, The Netherlands, June 2009.

[WL91] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel Distributed Systems*, 2(4):452–471, 1991.

List of publications

T. Bijlsma, M. J. G. Bekooij, and G. J. M. Smit. Circular buffers with multiple overlapping windows for cyclic task graphs. *Transactions on High-Performance Embedded Architectures and Compilers (HiPEAC)*, 2010. to appear.

T. Bijlsma, M. J. G. Bekooij, and G. J. M. Smit. Inter-task communication via overlapping read and write windows for deadlock-free execution of cyclic task graphs. In *Proc. Int'l Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, pages 140–148, Los Alamitos, CA, USA, July 2009. IEEE Press.

T. Bijlsma, M. J. G. Bekooij, P. G. Jansen, and G. J. M. Smit. Communication between nested loop programs via circular buffers in an embedded multiprocessor system. In *Proc. Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 33–42, New York, NY, USA, 2008. ACM Press.

T. Bijlsma, M. J. G. Bekooij, G. J. M. Smit, and P. G. Jansen. Efficient inter-task communication for nested loop programs on a multiprocessor system. In *Proc. Workshop of Circuits, System and Signal Processing (ProRISC)*, pages 122–127, 2007. Technology Foundation STW.

T. Bijlsma, M. J. G. Bekooij, G. J. M. Smit, and P. G. Jansen. Omphale: Streamlining the Communication for Jobs in a Multi Processor System on Chip. Technical report, Centre for Telematics and Information Technology, University of Twente, Enschede, 2007. Technical report TR-CTIT-07-44.

T. Bijlsma, and and P. G. Jansen. Energy Conservation with EDFI scheduling. In *Proc. European Conference on Smart Sensing and Context (EuroSSC)*, pages 259–261, Berlin/Heidelberg, October 2006. Springer.

T. Bijlsma, P. T. Wolkotte, and G. J. M. Smit. An Optimal Architecture for a DDC. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS), Reconfigurable Architecture Workshop (RAW)*, pages 192–200, Los Alamitos, CA, USA, April 2009. IEEE Computer Society.

S. J. Geuns, M. J. G. Bekooij, T. Bijlsma, and H. Corporaal, Parallelization of while loops in nested loop programs for shared-memory multiprocessor systems. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2011. to appear.