

MASTER

Parallel code generation for non-preemptively scheduled systems

John, S.

Award date:
2012

[Link to publication](#)

Disclaimer

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain



ELECTRONIC SYSTEMS, DEPARTMENT OF ELECTRICAL ENGINEERING

MASTER'S THESIS

PARALLEL CODE GENERATION FOR NON-PREEMPTIVELY SCHEDULED MULTIPROCESSOR SYSTEMS

Author:
Sunil John
0755704

Supervisors:
Prof. Dr. Henk Corporaal
Prof. Dr. ir. Marco Bekooij
ir. Stefan Geuns

November, 2012

Abstract

Streaming applications often have real-time constraints like throughput. In order to meet these real-time constraints, task-level parallelism is exploited. However, manual partitioning of sequential code to parallel tasks is time-consuming and error prone. Therefore it is beneficial to automatically generate parallel code from a sequential description of the application, while guaranteeing the same functional behavior. The automatic parallelization tool Omphale, extracts a parallel task graph from a sequential nested loop program, which can then be executed on a multiprocessor platform. Non-preemptive scheduling is beneficial as it enables fast context-switching, and is essential in processors that do not support preemption. In this thesis, we introduce techniques in Omphale to generate code for a shared-memory multiprocessor platform, which uses non-preemptive scheduling and a communication library with buffers supporting only a single producer and consumer.

Streaming applications often contain *if* or *switch* statements. These statements can contain multiple assignment statements in different branches, writing to the same variable. After parallelization, such a variable corresponds to a buffer having multiple producers. Also, certain variables can be read multiple times. These variables correspond to buffers having multiple consumers. A method to transform such buffers having multiple producers and consumers, to buffers with only a single producer and consumer is introduced. Streaming applications can also contain arrays wherein the different elements in an array are read and written in a different order. The order in which a producer task writes data into a circular buffer, may not be the order in which the consumer task reads data from the buffer. This leads to a reordering problem. A modified sliding windows approach is presented to solve this issue, when only buffers with a single producer and consumer are supported.

In order to prevent race conditions between tasks and deadlock, synchronization is done by means of *acquire* and *release* statements. The *acquire* statements are blocking whereas the *release* statements are non-blocking. *Acquire* statements check for availability of space or data in a buffer. When a blocking *acquire* call in a task fails due to lack of space or data, it must return control back to the non-preemptive scheduler. On the next execution of this task, it begins execution from the first statement in the task and not at the point at which it was blocked. This happens because the state of the task is not saved. It is shown that not saving the state of the task, can lead to erroneous functional behavior of the application and even a deadlock. Therefore, this thesis also introduces a method to save the state of the task by means of a finite state machine (FSM). Automatic parallelization approaches presented in literature, have not yet addressed these issues, when generating code for

multiprocessor systems using non-preemptive schedulers and blocking synchronization statements.

If variables are accessed across states in a FSM, they are stored statically in the task-state. This prevents the value from being lost, when control is returned to the scheduler on a blocking *acquire* call. Such variables potentially increase the memory usage, which is limited in an embedded system. The presence of a large number of states in the FSM leads to an increase in code size, and may lead to a decrease in throughput. Optimization techniques to reduce the number of states and the size of the task-state are developed. In order to reduce the number of states and task-state, *acquire* statements are moved or combined. Fine-grained synchronization on circular buffers can lead to a high synchronization overhead. By reordering the *acquire* calls in a task, the synchronization overhead in acquiring data in a buffer can be reduced. An attempt at a formal proof, showing that these transformations involving moving *acquire* statements over *release* statements or non-synchronization statements, and reordering *acquires* preserves deadlock-freedom, is presented.

A *Digital Video Broadcast - Terrestrial (DVB-T) Decoder* is used as a case-study. Parallel tasks are extracted from a sequential description of a DVB-T decoder. These parallel tasks are executed on the NXP MARS multiprocessor platform. By moving *acquire* calls, the number of states in the tasks of the DVB-T decoder are reduced by 60%. The size of the task-state is reduced by moving or combining *acquires*. The number of synchronization statements acquiring data in buffers is reduced by 30% by reordering the *acquire* calls in tasks.

Acknowledgements

I would like to express my gratitude for the people who offered me guidance and support during the course of my graduation project at NXP Semiconductors. Firstly, I wish to thank my supervisors, Prof. Henk Corporaal and Prof. Marco Bekooij for their guidance and constructive recommendations during the course of this project.

Prof. Henk Corporaal was my supervisor at Eindhoven University of Technology. He motivated me to think of alternate approaches at solving the various sub-problems in the project. I appreciate his valuable feedback to improve my presentations.

Prof. Marco Bekooij was my supervisor at NXP Semiconductors. I am grateful to him for offering the opportunity to work on this project. I would like to express my appreciation for the valuable time that he spent patiently, in explaining concepts and clarifying doubts. His insights in the subject played a role in guiding me towards interesting problems related to my project, that could be addressed.

I thank Dr. Pieter Cuijpers, for showing interest in my project and being part of the assessment committee.

I am deeply grateful to Stefan Geuns, who was also my supervisor at NXP Semiconductors. I appreciate the valuable feedback that he gave for this thesis, which has helped improve its quality. I thank him for all the guidance that he provided through the course of the project. I also wish to thank Joost Hausmans for his valuable time and efforts in clarifying doubts regarding related topics.

I wish to acknowledge the various suggestions provided by Umar Waqas during the course of the project and also for the thesis. I appreciate Sohan Walimbe for the inputs he provided during our discussions. I wish to thank my friends Siddharth Chunduri and Bhargava Puvvula for their help in proof-reading parts of the thesis. I am grateful to my friends from church, Jubin Jacob and Jonathan Vasu for their help and encouragement. Last, but not the least, I thank my parents and grandparents for their love and support. Above all, I thank and praise God Almighty, for His sustained grace and mercy, during the course of this project and in my life.

Contents

1	Introduction	1
1.1	Streaming Applications	1
1.2	Architecture	3
1.3	Automatic Parallelization	3
1.4	Problem Statement	4
1.5	Contributions	5
1.6	Outline	6
2	Omphale	7
2.1	Motivation	7
2.2	Tool Flow	8
2.2.1	Parallelization	8
2.3	Scope	10
3	Target Platform	13
3.1	Architecture	13
3.2	Communication/Scheduling Infrastructure	14
3.2.1	Components	15
4	Problem Description	17
4.1	Inter-task communication via circular buffers	17
4.1.1	Reordering Problem	18
4.1.2	Multiple Producers	18
4.1.3	Multiple Consumers	19
4.2	Non-Preemptive Scheduling	20
5	Inter-Task Communication via Circular Buffers	23
5.1	Reordering Problem	23
5.1.1	Related Work	23

5.1.2	Proposed Solution	25
5.2	Multiple Producers	27
5.2.1	Related Work	28
5.2.2	Proposed Solution	29
5.2.3	Implementation in Omphale	31
5.3	Multiple Consumers	32
5.3.1	Related Work	32
5.3.2	Proposed Solution	33
5.3.3	Implementation in Omphale	33
6	Non-Preemptive Scheduling	35
6.1	Infinite While Loop	37
6.2	Conditional While Loop	38
6.3	Nested While Loop	39
6.4	For Loop	40
6.5	Function Statement	41
6.6	Assignment Statement	41
6.7	If-Else Statement	41
6.8	Nested If-Else Statements	43
7	Optimizations	47
7.1	Reduction of States in FSM of tasks	47
7.1.1	Moving Acquire/Release outside if-else statements	47
7.1.2	Combining Acquires	49
7.2	Reducing Task-State	56
7.3	Reducing Synchronization Overhead	57
7.3.1	Reordering Acquires	58
8	Case Study	63
9	Conclusion	71
	Bibliography	75

Introduction

In recent years, there has been a surge of embedded applications that employ digital signal processing. These range from miniature wireless sensors, cellular phones, to rockets being sent to space. A notable property of these applications is that they are centered around streams of data; involving real-time acquisition, processing and subsequent output of data. These applications are commonly referred to as streaming applications, which is the application domain under consideration in this work. This thesis is concerned with enhancement of the scope of the multiprocessor compiler, Omphale. It performs automatic parallelization of a sequential description of a real-time streaming application such that it can be executed on an embedded multiprocessor system. Omphale currently supports kernels with preemptive scheduling and a communication library which supports buffers with multiple producers and consumers.

In this thesis, the scope of Omphale is enhanced to target commercial kernels employing non-preemptive scheduling and buffers with only a single producer and consumer. The following section describes streaming applications in more detail. Section 1.2 describes an architecture onto which such streaming applications are mapped. Section 1.3 presents the motivation for automatic parallelization and how it is applicable to this work. Section 1.4 gives a brief description of the problems addressed in this thesis. Section 1.6 gives an overview of the remaining chapters in this thesis.

1.1 Streaming Applications

Streaming applications are characterized by an infinite stream of input data [1]. They can be found on embedded devices, desktops and servers with high computing power. They are applicable in the domains of digital signal processing, networking, encryption etc. More specifically, they are used in wireless baseband processing (DVB-T, DAB, DRM), medical image processing, sensor processing and phased array radar systems [2]. In the remainder of this thesis, references to streaming applications and stream processing applications are used interchangeably.

Streaming applications have certain properties [2] which characterize their behavior. They perform relatively simple local processing of a large amount of data. So usually, the energy costs for data communication dominate the energy cost of processing. Data arrives at the incoming nodes at a fixed rate. The communication bandwidth is application dependent and so a large variety in communication bandwidth is required. The size of the data items and the data rates are also application dependent.

Streaming applications exhibit input-data dependent behavior. They can contain if-statements or while-loops where the condition depends upon input data values from the stream. Also, the result of an index-expression used to access array elements may depend upon input data values. At compile time, the value of such a condition or index-expression cannot be evaluated. Such statements are referred to as *non-manifest* [3] statements.

These applications usually have strict real-time constraints such as throughput and end-to-end latency which have to be met. In addition, embedded devices have concerns like minimizing energy consumption to increase their battery life. Since the memory available on such devices is limited, the architecture on which such streaming applications are mapped, should be capable of handling these concerns.

A block diagram of a streaming application, namely a Digital Video Broadcast - Terrestrial (DVB-T) receiver, can be seen in Figure 1.1. As can be seen, a stream of data (RF signal) is obtained from an external source such as a wireless receiver. In each stage, the data items are processed for a limited amount of time. The independent processing stages can be executed on multiple processors in parallel.

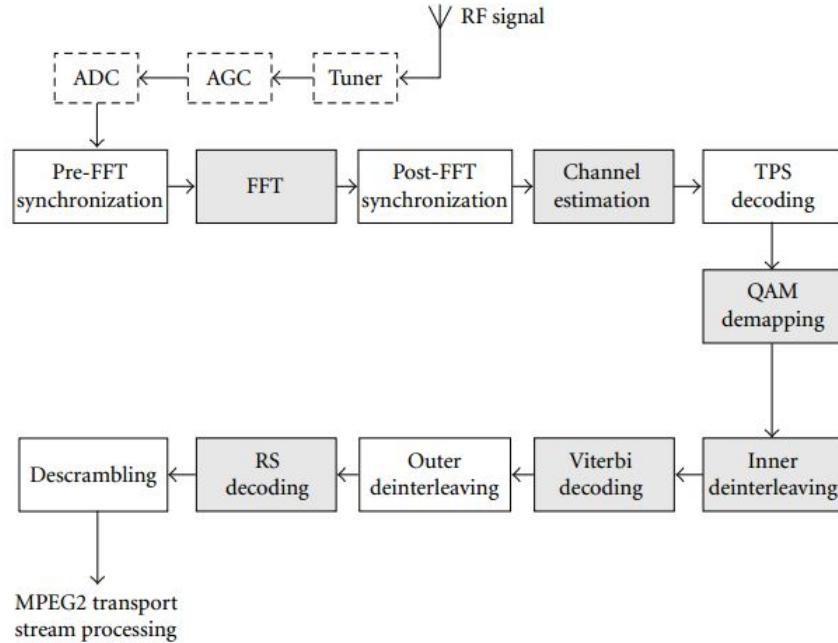


Figure 1.1: Example of a streaming application - DVB-T Receiver [4]

1.2 Architecture

A streaming application needs to be mapped on to a suitable architecture which can satisfy its performance requirements. This section describes an architecture which can meet these requirements. In order to meet the high throughput requirements of these applications on a single processor, it has to be clocked at a high frequency. However, this leads to high energy consumption, resulting in a low battery life. An alternative is to increase the number of processors, and reduce the frequency, thereby obtaining the same performance at a lower energy consumption. This has led to the use of multiprocessors in modern day systems. Most of the processor vendors are now moving towards multiprocessors [5]. As scaling technologies have improved, it facilitates the integration of multiprocessors on a single silicon chip, known as a Multiprocessor System-on-Chip (MPSoC). This aids in reducing the chip area. Most embedded systems have requirements of high computational power and low power consumption, which can be met efficiently by using MPSoCs. They are widely used in networking, communications, signal processing and multimedia applications [6]. At NXP, such an MPSoC namely, Multi Application Radio System (MARS) has been developed, targeting Software Defined Radio (SDR) applications. It has been chosen as the target platform to be used in this work. The MARS MPSoC has been described in detail in Chapter 3.

The tasks running on an MPSoC have to be scheduled for execution on the multiple processors. Non-preemptive scheduling is beneficial as it enables fast context-switching as compared to preemptive schedulers. It is essential if the processors do not support preemption. It is desired that our automatic parallelization tool Omphale, is able to support non-preemptive scheduling. Therefore, we consider an architecture which employs non-preemptive scheduling.

An MPSoC has a lot of potential in terms of the parallel processing that it can offer. However, in order to utilize it efficiently, the application has to be parallelized. This means that the following issues have to be solved to map the application on the available resources [7],

1. Dividing a program consisting of a single task into one with multiple balanced and communicating tasks. This process should also take the properties of the target platform into account.
2. Managing inter-task communication and synchronization. This needs to be considered carefully in order to avoid serialization of the parallel program and the occurrence of race conditions. Furthermore, it can introduce the risk of deadlocks.

Solving these issues manually is a time-consuming, labor-intensive and error-prone process. Therefore it is beneficial to address these issues through automatic parallelization.

1.3 Automatic Parallelization

Automatic parallelization involves automated partitioning of a sequential application to parallel tasks. This section motivates the need for automatic parallelization and how it is beneficial to include it in a multiprocessor compiler. A multiprocessor compiler

creates an executable of an application, which can be executed on a multiprocessor system. It maps parallel tasks onto the processors of the MPSoC in such a way that the temporal requirements of the application are met. It also ensures that inter-task communication and synchronization are handled correctly.

As described in section 1.1, streaming applications exhibit stages while processing the streaming data. This renders them a suitable candidate for function or task-level parallelism (TLP). These applications are usually described in sequential programming languages such as C and C++. In order to execute such an application on an MPSoC, it has to be partitioned into tasks that can run in parallel on the MPSoC.

A multiprocessor compiler maps the parallel tasks onto different processors of an MPSoC. It could obtain the parallel partitioned tasks of the application as input or obtain a sequential description of the application and perform the partitioning itself. If the partitioned tasks are obtained as input, it is difficult to guarantee deadlock freedom. In this case, the process of partitioning and inserting appropriate communication and synchronization statements would have been done manually by the system designer or application developer. In this approach, it is also possible to do optimizations manually and exploit the architecture available. Also, the granularity of the tasks can be decided in an optimized manner [3]. These optimizations require the system designer or application developer to have a thorough understanding of the application. The downside of this approach is that it is time-consuming and error-prone. Thereby, it will increase the time-to-market of the application.

On the other hand, if a sequential description is taken as an input, the partitioning of tasks can be done by the compiler. By starting with a sequential description of the application, the advantage is that it is deadlock-free, deterministic and free of race conditions. The input description of the application should be in a manner that the data dependencies in the application are analyzable [3]. This would enable the partitioning of the application into parallel tasks. Also, the inter-task communication and synchronization statements can be inserted automatically by the compiler, such that there is no deadlock. Deadlock-freedom can be checked if the multiprocessor compiler is able to generate a suitable model of the input application which supports this check. This also enables temporal analysis of the application for throughput requirements.

The multiprocessor compiler Omphale, is a research tool currently under development by PhD students at NXP. Given a sequential description of a streaming application, it carries out automatic parallelization, generates an analysis model and produces an executable for a target multiprocessor system. The aim of this work is to enhance the scope of Omphale to be able to target a wider class of multiprocessor systems with non-preemptive scheduling and communication libraries having buffers with only a single producer and consumer. This is further discussed in the next section.

1.4 Problem Statement

The problems addressed in this thesis are described briefly in this section. A detailed problem description is given in Chapter 4.

The automatic parallelization tool Omphale generates parallel code for a multiprocessor system, from a sequential description of a streaming application. However, it currently supports multiprocessor systems with only preemptive schedulers and a communication library built in-house, with buffers supporting

multiple producers and multiple consumers. It is desired that Omphale can target the NXP MARS multiprocessor platform. The problem is that MARS uses a non-preemptive scheduler and a communication library having buffers, with only a single producer and consumer, which are not supported by Omphale. Each of these sub-problems is described briefly in the following paragraphs.

Non-preemptive scheduling is beneficial as it enables fast context-switching in comparison to preemptive schedulers. It is essential if the processors do not support preemption. However, Omphale does not support non-preemptive schedulers. When a non-preemptive scheduler is used and a task blocks due to a blocking synchronization call, it must return control back to the scheduler to prevent indefinite blocking and allow other tasks to execute. When the task executes again, it starts from the beginning of the task and not at the point, where it had yielded control back to the scheduler. This could lead to erroneous functional behavior of the application and even deadlock.

The sequential description of a streaming application taken as input by Omphale, can contain multiple assignment statements writing to a single variable. It can also contain multiple statements reading from a variable. These would correspond to buffers with multiple producers and consumers. However, the communication library used on MARS does not support such buffers. It only supports buffers with a single producer and consumer. Omphale is unable to handle such input applications to generate code for multiprocessor systems, which use a communication library with buffers having only a single producer and consumer.

1.5 Contributions

This project aims at augmenting the usability of the multiprocessor compiler Omphale, to target multiprocessor systems utilizing communication and scheduling frameworks with certain properties. These properties include the use of circular buffers having only a single producer and consumer and non-preemptive scheduling. The Multi Application Radio System (MARS) MPSoC, developed at NXP, is the chosen target platform which utilizes such a communication and scheduling framework. The contributions of this thesis are as follows.

- Developed a wrapper synchronization library, containing a modified sliding windows buffer implementation, to handle circular buffers with producers and consumers having different access patterns in an array.
- Developed methods in Omphale to transform circular buffers with multiple producers and consumers to buffers with only a single producer and consumer
- Developed a mechanism to save the state of the task in non-preemptively scheduled multiprocessor systems, to prevent erroneous functional behavior of an application when using blocking synchronization statements
- Proposed optimizations to reduce the number of states in the generated FSMs in tasks, the size of the task-state and the number of synchronization statements.
- Implemented the proposed solutions in Omphale, and evaluated them on the MARS multiprocessor platform using DVB-T decoder as a case-study.

1.6 Outline

An outline of the remainder of the thesis is presented in this section. Chapter 2 describes the tool flow used in Omphale, starting from a sequential description of a streaming application, and obtaining an executable as an output. Chapter 3 highlights the features of the NXP MARS MPSoC which is used as the target platform. Chapter 4 presents a detailed problem description. Chapter 5 describes the proposed solutions to the issues concerned with inter-task communication via circular buffers. These issues include the reordering problem and the handling of buffers with multiple producers and multiple consumers. In Chapter 6, the problem of explicit state management for tasks when using non-preemptive scheduling and means of solving it are discussed. In Chapter 7, optimizations are proposed to reduce the static memory requirements in tasks and improve the performance of streaming applications on a target architecture. A case-study which illustrates the use of the proposed solutions is presented in Chapter 8. Finally, Chapter 9 concludes the thesis and presents future work.

Omphale

A multiprocessor compiler takes a streaming application as input and produces an executable which can run on a multiprocessor system. In this chapter, the multiprocessor compiler Omphale is described in detail. Omphale, currently under development at NXP, takes a sequential description of a streaming application along with its real-time constraints such as throughput and produces an executable that can be executed on a target multiprocessor system. The advantages of starting with a sequential description of the streaming application have been highlighted in section 1.3. The various phases that are followed in transforming a sequential streaming application to a parallel task graph, which is executable on a multiprocessor system, are described in this chapter.

2.1 Motivation

This section provides the motivation for using Omphale over other multiprocessor compilers. By starting with a sequential description of a streaming application, it relieves the programmer of the burden of manual insertion of communication and synchronization statements in the parallel tasks. The semantics of Omphale Input Language (OIL) are such that the data dependencies can be extracted [3]. Omphale also supports non-manifest statements in the NLP, which correspond to input-data dependent behavior. A parallel task graph can be extracted by analyzing the data dependencies. A task graph comprises of a dependency graph with inter-task communication and synchronization statements inserted into the tasks. A dataflow model is extracted from the input streaming application such that temporal requirements can be analyzed. Thereby, on providing the temporal requirements of the application as input, sufficient buffer capacities can be computed if the temporal requirements can be met.

2.2 Tool Flow

Omphale takes a sequential description of a streaming application as input. This is expressed as a nested loop program (NLP) in OIL. The temporal constraints of the application are also provided as input along with the NLP using OIL. The compiler operates on these inputs and produces an executable targeting a multiprocessor system as an output. This process is carried out in three phases [3], as shown in Figure 2.1. In the parallelization phase, the data dependencies in the input NLP are analyzed to create a parallel task graph. A dataflow model, corresponding to this task graph is extracted in this phase. This dataflow model is used to calculate sufficient buffer capacities such that the temporal requirements of the application are met [8]. Following this, the resource allocation phase assigns the parallel tasks to processors. Buffers are then allocated in memory according to the buffer sizes obtained from the parallelization phase. In the linking phase, the parallel tasks are linked with the communication library and the kernel. In the following subsection, a detailed description of the parallelization phase is furnished.

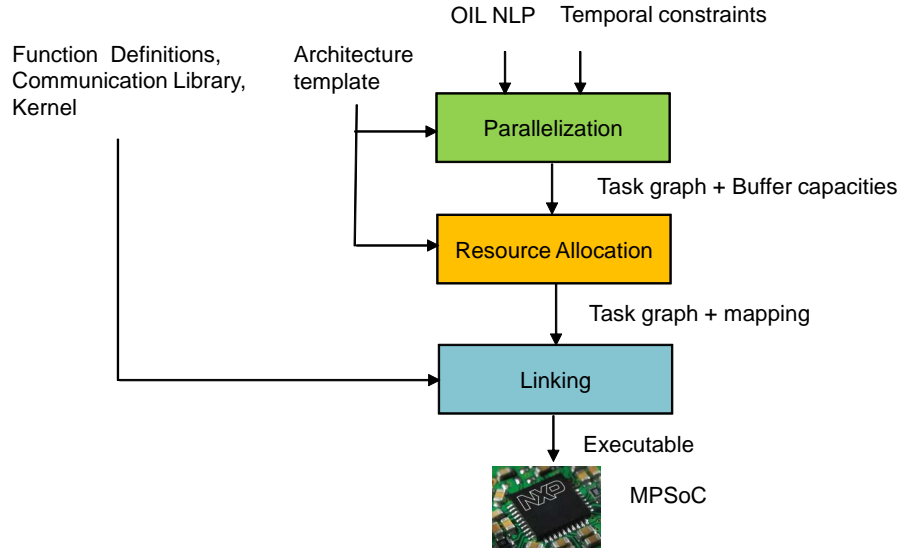


Figure 2.1: Omphale Tool Flow

2.2.1 Parallelization

In the parallelization phase, a task graph along with its corresponding Cyclo-static Synchronous Dataflow (CSDF) graph are extracted from the input NLP. This process is carried out in three sub-phases [3]. In the first sub-phase, a dependency graph is extracted from the input NLP, highlighting the data dependencies between tasks. Then inter-task communication and synchronization statements are inserted into the tasks, thereby forming a task graph in the second sub-phase. Subsequently, based on the inserted synchronization statements, a CSDF model is extracted in the third sub-phase. The first sub-phase is described in more detail. For more details about the other sub-phases, the reader is referred to [3].

2.2.1.1 Extraction of Dependency Graph

The extraction of a dependency graph requires the identification of tasks that can be executed in parallel. The assignment statements and function calls in the input NLP are made into parallel tasks. In case of sequential execution, the order of execution of the statements in the NLP guarantees functional correctness. After parallelization, the dependencies have to be maintained, to guarantee the same functional behavior as the sequential application. The data dependencies can be found if the statements in the NLP satisfy *single assignment* (SA) [9].

Single Assignment Single assignment means that a scalar or a element in an array is written only once. There are two forms of single assignment, static single assignment (SSA) and dynamic single assignment (DSA). SSA [10] requires that there is at most one statement that writes to a scalar or array. DSA requires that a scalar or an element of an array is written only once during the entire execution of a program [11]. A program that satisfies SSA need not satisfy DSA or vice-versa.

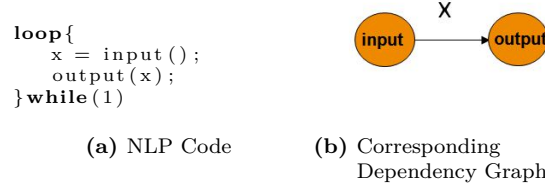
For while loops with an unknown iteration bound, both SSA and DSA may not be satisfied [12]. SSA requires that a variable is written only once. However, in streaming applications, there can be several writes to the same variable in the branches of an *if* statement or a *switch* statement. Therefore, SSA is not satisfied. In different iterations of a while loop, a variable may be written more than once. Therefore DSA is not satisfied. If automatic renaming is done, and a corresponding array is made to store the values of every iteration, it would require an array of infinite size. Hence a new form of single assignment, namely single assignment section (SAS) [12] is introduced. During the execution of a SAS, each scalar and array element can be written at most once. In the NLP, multiple SASs may exist and each scalar or array can have multiple SASs. At the end of a SAS, the value of the variable in the SAS is lost. The reader is referred to [13] for further description about SASs.

Omphale Input Language (OIL) In order to extract a dependency graph, the data dependencies in the input NLP must be analyzed. To analyze the data dependencies, the input language OIL requires that they must be made explicit. Therefore OIL does not support pointers. An NLP expressed in OIL must also satisfy the notion of single assignment specified by SAS. For the first iteration of a while loop, the SAS also includes the statements prior to the loop in the NLP.

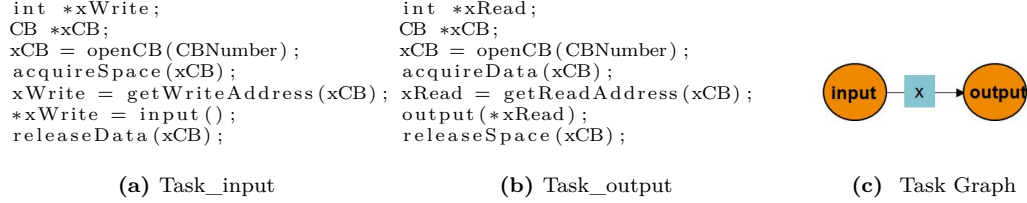
OIL permits the use of external C-functions thereby enabling re-use of existing function implementations. The requirement is that these C-functions should be side-effect free [3]. This implies that the functions do not have any implicit data dependencies. OIL supports the use of the following statements in a NLP, namely assignments, functions, if-else, while loops and for-loops. It also supports non-manifest if statements and while loops. For the examples shown in this thesis, the symbol '~' is used to represent code that has been omitted for clarity. It also includes code that is non-manifest.

An example of the extraction of a dependency graph is shown in Figure 2.2. An example NLP is shown in Figure 2.2a, where a variable *x* is the data dependency between two functions *input* and *output*. In the corresponding dependency graph in Figure 2.2b, it is illustrated how assignment statements and function calls become tasks, sharing the data dependency.

After a dependency graph has been extracted, inter-task communication and

**Figure 2.2:** Extraction of Dependency Graph

synchronization statements have to be inserted into the tasks. Here the shared variables (data dependencies) between tasks are replaced by circular buffers. Figure 2.3c shows the task graph corresponding to the dependency graph in Figure 2.2. Here, the variable x has been replaced by a circular buffer. The inter-task communication and synchronization statements for the *input* and *output* tasks are inserted as shown in Figure 2.3a and Figure 2.3b respectively. The communication statements pertain to obtaining a reference to the circular buffer using *getWriteAddressCB* and writing to it. The synchronization statements *acquireSpace* and *acquireData* are used to check the availability of space and data respectively in the buffer. Similarly *releaseData* indicates that the data has been written to and is available for reading, and *releaseSpace* indicates that data has been read and is now available for writing.

**Figure 2.3:** Extraction of Task Graph

An example of a channel decoding application and its corresponding task graph are shown in Figure 2.4. From the NLP code in Figure 2.4a, it can be seen that the application can either be in the acquisition (acq) phase or the decoding phase depending on the variable *state*. The decode phase can have several stages forming a pipeline, as shown by the functions *decode1*, *decode2* and *output*. The keyword *out* is used to represent the output of a function. In this example, *out state* refers to a write to the variable *state* in the next iteration of the while loop. From Figure 2.4a, it can be seen that circular buffers can have multiple producers and multiple consumers. The buffer x has multiple consumers and the buffer *state* has multiple producers and consumers. The fact that multiple producers are mutually exclusive is no longer visible in the task graph, but can only be seen in the input NLP.

2.3 Scope

In the multiprocessor platforms that Omphale currently supports, the communication library and the kernel have been developed in-house for the target platform. In being

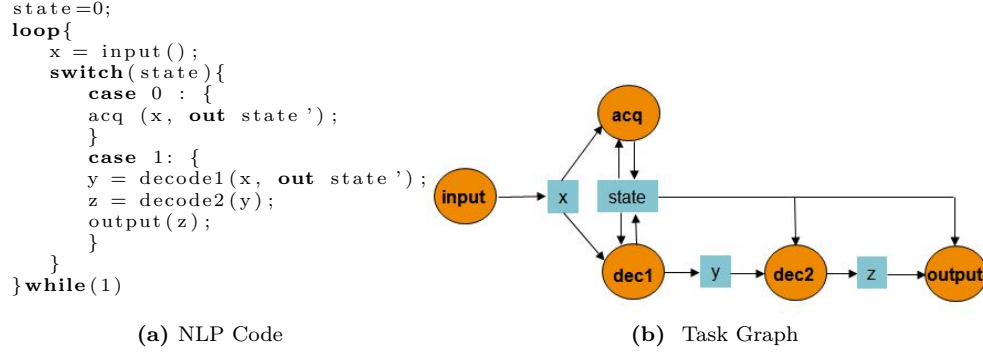


Figure 2.4: Extraction of Task Graph

able to use commercial kernels and communication libraries, a wrapper has to be generated around the commercial communication library so as to support the required communication and synchronization behavior. Such a wrapper generation is essential in the work carried out in this project. The kernels supported currently by Omphale are developed in-house and they use preemptive schedulers such as Time Division Multiplexing (TDM) schedulers, which belong to the category of budget schedulers. Budget schedulers provide tasks with a minimum budget for every shared resource such that real-time guarantees can be given.

Target Platform

The Multi Application Radio System(MARS) MPSoC, developed at NXP, is the target platform used in this work. It uses the Sea-of-DSP (SoD) infrastructure which has a streaming kernel with non-preemptive scheduling. The communication library offered by SoD supports only buffers with a single producer and consumer. MARS facilitates the implementation of software defined radios (SDR).

Software Defined Radio

A software-defined radio system, or SDR, is a radio communication system where components that have been typically implemented in hardware like mixers, filters, amplifiers, modem and detectors are instead implemented by means of software on a personal computer or embedded system [14].

The processing stages in a typical SDR transceiver [15] are shown in Figure 3.1. In the case of an SDR receiver, the first stage is the RF block. The incoming signal is sampled by an analog-to-digital (ADC) converter. The digitized signal is passed to the baseband processing section which consists of the digital front end (DFE), Modem and Codec blocks. The output from this section is passed to the application layer for further processing. The digital front-end consists of filters that are used to remove the unwanted frequencies and suppress the noise present in the signal. The modem block performs the demodulation followed by decoding, which is done by the codec block. In the case of a transmitter, the data flows in the reverse direction starting from the application layer towards the transmitter RF block.

3.1 Architecture

MARS facilitates the implementation of SDRs. The architecture of the MARS platform is shown in Figure 3.2. It consists of a Digital Front End (DFE) which receives the data provided by the transmitter. There are two Vector Digital Signal Processors (VDSPs) which execute the filtering and demodulation operations in the baseband processing. A

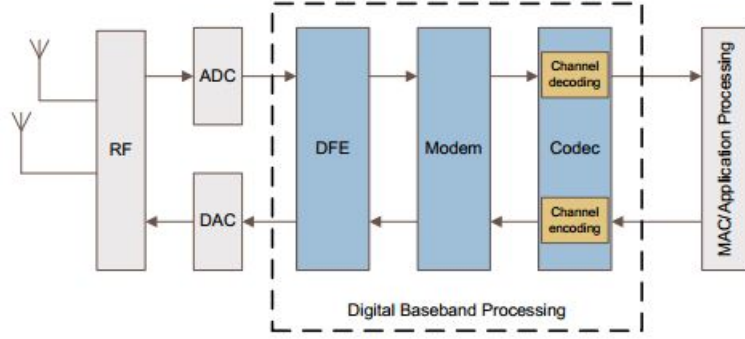


Figure 3.1: SDR Transceiver [16]

hardware accelerator, FLORA implements the decoding functions. In order to configure the VDSPs and the FLORA, a general purpose processor namely the ARM is used. There are two Static random access memories (SRAMs) to store data. An external SDRAM is also present in case storage space is not sufficient in the other memories. The various processors and memories in the MARS are interconnected by a high speed Arm Extensible Interface (AXI) bus. The processed output data from the MARS is available to a host system through a USB connection.

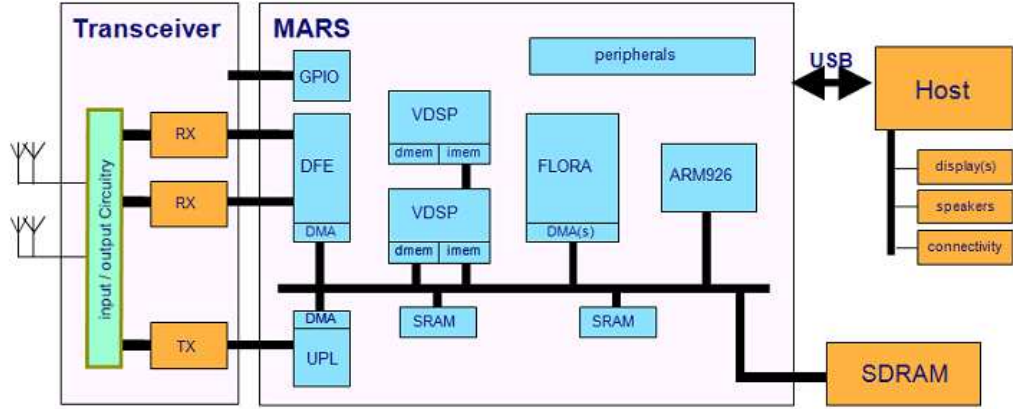


Figure 3.2: MARS

3.2 Communication/Scheduling Infrastructure

Sea-of-DSP (SoD) is the communication and scheduling infrastructure used by the ARM and VDSPs in the MARS platform. It facilitates the creation and modification of signal processing task graphs during run-time of the application. It efficiently dispatches the signal processing tasks onto the different processors.

3.2.1 Components

To facilitate creation, modification and dispatch of the signal processing tasks on different processors, SoD has two components namely the network manager and a streaming kernel. Figure 3.3 shows the block diagram of the components in SoD and their presence on the different processors. The network manager is present on the general purpose CPU, which is ARM in case of the MARS. The streaming kernels are present on all the processors and manage the function tasks that are executed on these processors. The DSP hardware corresponds to the VDSPs present in the MARS. The functions of these SoD components are described below.

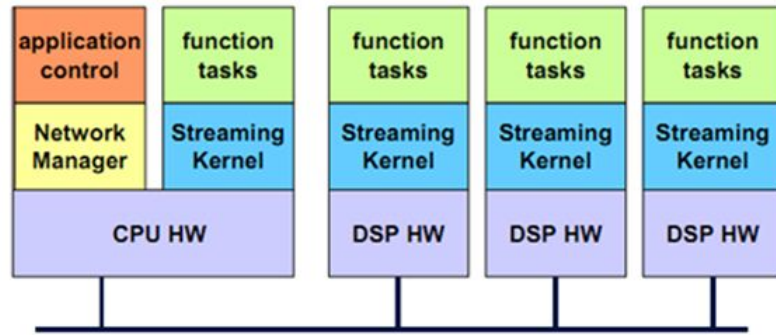


Figure 3.3: Sea-of-DSP (SoD)

3.2.1.1 Network Manager

The Network Manager provides an API for managing the signal processing tasks running on different processors. It provides the following facilities to the user.

- Create, delete processing tasks
- Setting up task graph by connecting or disconnecting tasks via communication channels
- Suspend, resume tasks
- Exchange of commands and status information with tasks

3.2.1.2 Streaming Kernel

The streaming kernel is responsible for dispatching the signal processing tasks onto the various processors. It uses a non-preemptive round robin scheduler. It executes the task schedule and supports the data communication required by the tasks. The data exchange between tasks is done through software buffers. It provides a communication library with API calls to facilitate exchange of data between tasks.

Communication Library The communication library provides buffers and API calls to check the availability of data or space in the buffer and to read/write data from/to

the buffer. Every task has a port which is its point of access to a channel. A single task may have multiple ports. A unidirectional channel is created by connecting two ports of tasks together. Every channel can have only a single producer and consumer.

Problem Description

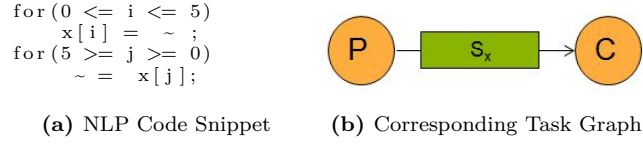
This thesis is concerned with the enhancement of the scope of the multiprocessor compiler Omphale, which performs automatic parallelization. Omphale takes a sequential description of a real-time streaming application as input and parallelizes it, such that it can be executed on an embedded multiprocessor system. Omphale currently supports kernels with preemptive scheduling and a communication library with buffers supporting multiple producers and consumers. The scope of Omphale is now enhanced to target commercial kernels employing non-preemptive scheduling and buffers with only a single producer and consumer. The MARS multiprocessor platform, developed at NXP, is a target platform which uses such a commercial kernel. A parallel task graph of a stream processing application needs to be generated such that it can be executed on MARS.

4.1 Inter-task communication via circular buffers

Motivation

Parallel tasks are extracted from a sequential description of streaming application expressed in OIL [3][12]. These tasks communicate via circular buffers. As shown previously in Figure 2.4, these circular buffers can have multiple producers and consumers. Also, in the case of reading and writing to arrays, the producer task could write to the circular buffer corresponding to the array in a particular order and the consumer task could read from the buffer in a different order. The array index could also be non-manifest, i.e. it is dependent on the input data and therefore not known at compile time. Therefore the order of access in the array is not known at compile time.

In the above cases, the communication library on the multiprocessor system should be capable of handling buffers with multiple producers and consumers and also provide a means for out-of-order accesses for arrays. However, the communication library used on our target platform MARS, supports only buffers with a single producer and consumer. A possible approach to solving this incompatibility is to change the existing communication library and replace it with a new one. This would be challenging as

**Figure 4.1:** Illustration of Reordering Problem

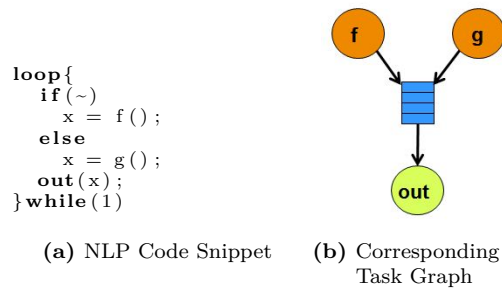
applications using these libraries would contain legacy code and would require a lot of effort to change it. The alternative is to build a wrapper around the existing library. This is the approach followed in this work. Moreover, building a wrapper makes it easier to target other commercial kernels which have their own communication libraries. The following subsections discuss each of the sub-problems in more detail.

4.1.1 Reordering Problem

The reordering problem occurs when a producer and consumer of an array have different access patterns, when accessing a circular buffer. This is illustrated with an example shown in Figure 4.1. Figure 4.1a shows a code snippet of a nested loop program (NLP). Here it can be seen that the array x is written to inside the first *for* loop and corresponds to the producer task (P). The array x is read inside the second *for* loop and this corresponds to the consumer task (C). The corresponding task graph showing the data dependencies between the two tasks is shown in figure 4.1b. It can be seen that the first element that the producer task would write to the buffer is the array index of 0, whereas the consumer task reads an array index of 5.

4.1.2 Multiple Producers

The multiple producers problem is encountered when there are two tasks which write to the same buffer. OIL supports *if* and *switch* statements which leads to the possibility of having multiple mutual exclusive producers to a buffer. The problem is illustrated with an example shown in Figure 4.2. It can be seen from the code snippet of the NLP in Figure 4.2a that there is a mutual exclusive write to the variable x , depending on the *if* condition. After parallelization, each of the reading and writing statements correspond to different tasks as seen in Figure 4.2b. The producer tasks f and g write to the buffer and the consumer task out reads from the buffer. The issues of concern are listed below.

**Figure 4.2:** Illustration of Multiple Producers Problem

- The communication library in the target platform does not support buffers with multiple producers. Buffers having multiple producers should be transformed such that buffers with a single producer and consumer are used.
- On using buffers with a single producer and consumer, the consumer task is unable to identify which of the producers wrote to its corresponding buffer. As can be seen from Figure 4.3, task *out* is unable to identify whether task *f* or task *g* wrote to their corresponding buffers.

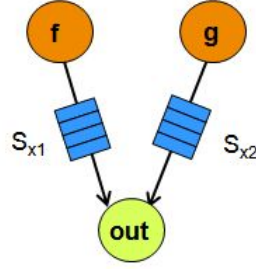


Figure 4.3: Multiple Producers Problem - Separate buffers

4.1.3 Multiple Consumers

The problem with multiple consumers occurs when there are two tasks which need to read from the same buffer. This is illustrated in the example in Figure 4.4. It can be seen from the code snippet of the NLP in Figure 4.4a that the variable *x* is written once and read twice. The second read of the variable *x* is a conditional read. After parallelization, each of the reading and writing statements correspond to different tasks as seen in Figure 4.4b. The producer task *f* writes to the buffer and the consumer tasks *out1* and *out2* read from the buffer. The issues of concern are listed below.

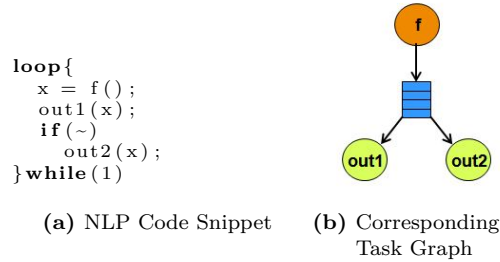


Figure 4.4: Illustration of Multiple Consumers Problem

- The communication library in the target platform does not support buffers with multiple consumers. Buffers having multiple producers should be transformed such that buffers with a single producer and consumer are used.
- On using buffers with a single producer and consumer, if at least one of the tasks conditionally reads its corresponding buffer, there is an accumulation of data in

that buffer. This would lead to the need for infinite size buffers. This is illustrated in Figure 4.5. The task f writes data to the buffers corresponding to the producer-consumer pairs $(f, out1)$ and $(f, out2)$. However, if the condition in task $out2$ is not satisfied, it would not read from its corresponding buffer. Therefore, there would be an accumulation of data in this buffer, which means an infinite size buffer is required.

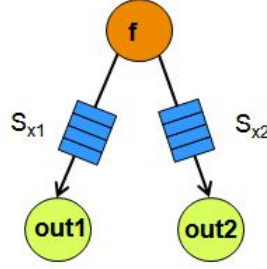


Figure 4.5: Multiple Consumers Problem - Separate buffers

4.2 Non-Preemptive Scheduling

Omphale currently supports preemptive schedulers. It is desired to extend Omphale to support non-preemptive schedulers. Non-preemptive schedulers enable fast context-switching and are needed in cases when the processors do not support preemption.

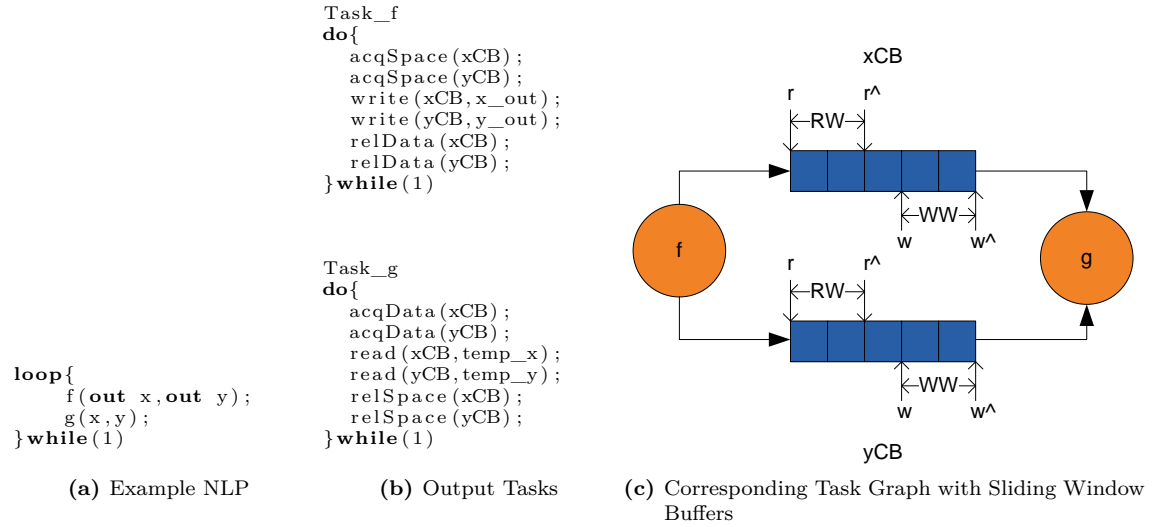
When a non-preemptive scheduler is used and any of the generated parallel tasks blocks on encountering a blocking synchronization call, the task returns control back to the scheduler so that other tasks can execute. When the task is executed again, it executes from the beginning because the state of the task was not saved. This could lead to an erroneous functional behavior of the application.

The blocking calls found in the generated tasks are the *acquire* statements namely, *acqSpace* and *acqData* which check for the availability of space or data respectively in a circular buffer. When there is no space or data available in the buffer, these calls return control back to the scheduler. Consider a task containing an *acqData* statement followed by another *acquire* statement, with a *releaseSpace* in between. If this task blocks on execution of the second *acquire* statement, the data corresponding to the first *acquire* statement, is already removed from the buffer. This data cannot be obtained on the next execution of the task. This is illustrated in the example shown in Figure 4.6. The input NLP is shown in Figure 4.6a. After parallelization, two tasks, namely *task_f* and *task_g* are generated. A snippet of *task_g* is shown in Figure 4.6b. In *task_g*, the value of x is read into a variable *condition*. On executing *task_g*, if *condition* is true and the *acqSpace* call fails due to lack of space, then *task_g* returns control back to the scheduler. On the next execution of *task_g*, it executes from the beginning and not at the point where the task had got blocked. When *task_g* executes from the beginning, a new value of x is read from xCB and the function $g()$ is conditionally executed based on this value of x . This is erroneous as the execution of the function $g()$ should depend on the previous value of x which was read from xCB . However, the previous value of x was lost as it was removed from the buffer.

<pre> loop{ x=f(); if(x) y=g(); }while(1) </pre> <p>(a) Example NLP</p>	<pre> do{ acqData(xCB); read(xCB,condition); relSpace(xCB); if(condition){ acqSpace(yCB); write(yCB,g()); relData(yCB); } else { acqSpace(yCB); relData(yCB); } }while(1) </pre> <p>(b) Task_g corresponding to the statement y=g() in the example NLP</p>
---	--

Figure 4.6: Non-Preemptive Scheduling

If the state of the task is not saved, it could even lead to a deadlock. This is illustrated by an example in Figure 4.7. The input NLP shown in Figure 4.7a consists of two functions f and g , with two data dependencies between them. After parallelization, two tasks $task_f$ and $task_g$ are obtained and their corresponding task graph is shown in Figure 4.7c. When sliding window [3] buffers are used for inter-task communication, a read window (RW) and a write window (WW) determine which locations of the buffers are accessible. The read window is defined by r^{\wedge} at the head and r at the tail. Similarly, a write window is defined by w^{\wedge} at the head and w at the tail. A snippet of the code in $task_f$ and $task_g$ are shown in Figure 4.7b. When $acqData(xCB)$ is called in $task_g$, if there is data in the location after r^{\wedge} , r^{\wedge} is moved forward by one location. When executing $task_g$, if $acqData(xCB)$ succeeds and $acqData(yCB)$ fails, then $task_g$ is blocked and returns control back to the scheduler. On the subsequent executions of $task_g$, if the same scenario repeats, then r^{\wedge} coincides with w and $task_g$ cannot proceed as there is no new data written that is available for reading. Similarly, w^{\wedge} coincides with r and cannot proceed. This leads to a deadlock as the read window and the write window occupy the entire buffer and both cannot advance. The deadlock occurs as some locations in the buffer are acquired by the $acqData$ statement but not released by the corresponding $relSpace$ statement. This occurs because on subsequent executions of the task after being blocked, it resumes execution from the beginning and not from the point where it had got blocked.

**Figure 4.7:** Illustration of Deadlock with Non-preemptive scheduling

Inter-Task Communication via Circular Buffers

In this chapter, the various issues in dealing with inter-task communication via circular buffers and the solutions proposed to solve them are discussed. The solutions proposed in literature in dealing with these issues are described in corresponding sections. Section 5.1 discusses the reordering problem and the generation of a wrapper synchronization library. In Section 5.2, the multiple producers problem and proposed solutions are described. Section 5.3 contains a description about the issues with multiple consumers and proposed solutions to solve them.

5.1 Reordering Problem

The producer and consumer of an array can have different access patterns while accessing the array, thereby leading to the reordering problem. A sequential NLP can contain statements reading and writing to an array, which after parallelization, correspond to parallel tasks reading and writing to a buffer corresponding to the array. It is noteworthy to mention that if the order in which the different elements in the array are read and written is the same, then the reordering problem does not occur. This is because the elements of the array are written and read in FIFO order and the FIFO buffer present in the communication library supports this behavior.

5.1.1 Related Work

To solve the reordering problem, FIFO buffers in combination with a reordering task and reordering memory can be used [3]. This is illustrated in Figure 5.1. The producer task writes the values in the order it produces, to a FIFO buffer. Then the reordering task which has the knowledge of both the write access pattern and the read access pattern of the array writes them to a reordering memory. From the reordering memory, the reordering task writes to another FIFO buffer following the read access pattern of the array. In this case, the producer task and the reordering task execute sequentially as the elements of the entire array have to be written into the reordering memory before the

reordering task can reorder them. This approach has the disadvantage that it prevents parallel execution of both the tasks. Also there is the overhead of the reordering task and reordering memory. Furthermore, two FIFO buffers are required.

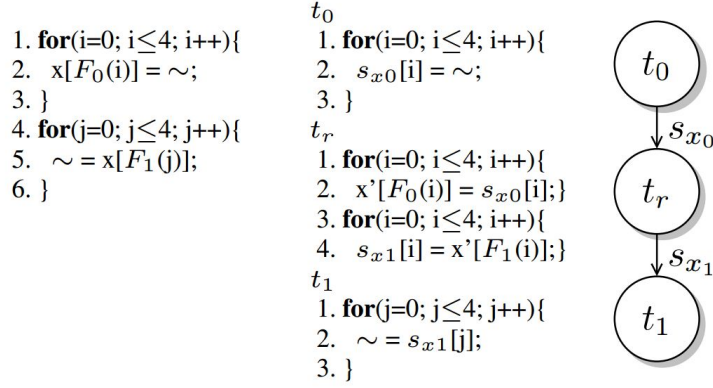


Figure 5.1: Reordering task with reordering memory [3]

Another method using FIFO buffers has been proposed in [17] by Turjan et al. A special controller unit is designed that takes care of restoring the order of the tokens. Also additional memory is required in the consumer task. Though in this case, no reordering task is required, it faces the overhead of a complex controller unit and also reading from the reordering memory. In [18], Huang et al. propose the concept of a windowed FIFO. A windowed FIFO requires local buffers or containers in the reading and writing tasks. Dedicated logic is used to control the writing of elements of the array into the local container on the producer side, after which the contents of the container are transferred to the FIFO between the producer and consumer tasks. Similarly, dedicated logic is used to control the writing to the container on the consumer side, after which the consumer task can read from it. This has been illustrated in Figure 5.2. In this approach, there is the overhead of the local containers which can be likened to reordering memory.

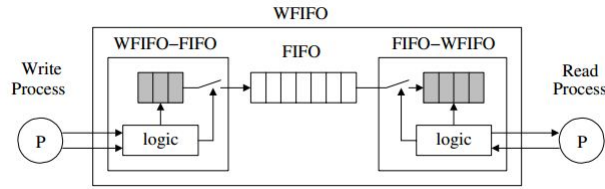


Figure 5.2: Windowed FIFO [18]

A buffer which supports out-of-order access can also be used to solve the reordering problem. Such a buffer enables reading and writing to any location in the buffer. Synchronization is needed to prevent locations in the buffer being read, before they have been written [3]. Bijlsma et al. developed two such buffers, namely *Sliding windows* [19] and *Overlapping windows* [20]. Sliding windows consist of a read window for every reading task and a write window for every writing task. The read windows are not allowed to overlap with the write windows. However, sliding windows cannot be applied always for cyclic data dependencies as it can cause a deadlock. Overlapping windows can be used even for latency critical cyclic data dependencies [3]. In the case of overlapping windows, a location is removed from the window as soon as it has been accessed.

5.1.2 Proposed Solution

The reordering problem occurs due to different access patterns in an array by the producer and consumer tasks. The solutions proposed in literature as shown in section 5.1.1 used FIFO buffers with the overhead of additional control logic and reordering memory. By using buffers permitting out of order access, the reordering problem was solved using sliding windows. The SoD communication library, used in our target platform permits out of order access. Therefore, the approach using sliding windows [20] seems applicable. However, due to certain issues which are described subsequently the sliding windows buffer is not directly applicable. Therefore, a modified version of sliding windows is developed.

Sliding windows consist of a read window and a write window in a circular buffer as shown in Figure 5.3. The read window (RW) is formed by the r^{\wedge} pointer at the head and the r pointer at the tail. Similarly, a write window (WW) is formed by the w^{\wedge} pointer at the head and the w pointer at the tail. These windows advance when a task calls the appropriate synchronization statements, which are described subsequently.

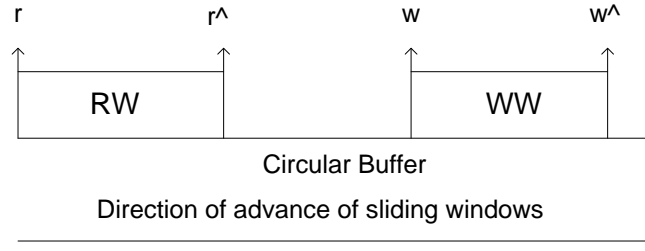


Figure 5.3: Sliding windows Buffer

Tasks executing on different processors and communicating via shared memory need a memory consistency model to prevent the reading of a location before it is written [3]. The memory consistency model defines the order in which write accesses complete. The sliding windows approach employs a streaming consistency model [21], wherein shared memory accesses can be reordered so that they can be pipelined for better performance. In order to prevent race conditions between the read and write accesses to shared memory, synchronization has to be performed. The synchronization ensures that a shared location is read only after it has been written. Similarly, synchronization ensures that a shared location has been read before it is overwritten. In the streaming consistency model, synchronization is performed using *acquire* and *release* statements.

An *acquire* statement precedes the access to a shared location and blocks until the location becomes available. A *release* statement is inserted after the access to a shared location, indicating that it is now available. An empty location in the buffer is referred to as *space* and a location filled with data is referred to as *data*. Before writing data to an empty location, a producer acquires it by a *acquireSpace* call. After data has been written to this location, the producer executes a *releaseData* call signaling that the location is now available for reading by the consumer. Subsequently, the consumer can acquire the location with a *acquireData* call. Once the data has been read, the consumer releases the location with a *releaseSpace* call.

The *acquire* and *release* synchronization statements advance the read and write windows in the buffer. The write window should always be ahead of the read window in order to prevent the reading of a location that has not been written. The advancement of the

windows by the producer and consumer are described below.

The following synchronization statements are employed by the producer task.

- ***acquireSpace***(*n*) increments w^{\wedge} by *n* locations
- ***releaseData***(*n*) increments *w* by *n* locations

The *acquireSpace* synchronization statement moves the head of WW forward by acquiring a space in the buffer. The corresponding *releaseData* call moves the tail of WW forward after having written a value in the corresponding location(s), implying that the data can now be read.

The following synchronization statements are employed by the consumer task.

- ***acquireData***(*n*) increments r^{\wedge} by *n* locations
- ***releaseSpace***(*n*) increments *r* by *n* locations

The *acquireData* synchronization statement moves the head of RW forward by acquiring *n* locations in the buffer. The corresponding *releaseSpace* call moves the tail of RW forward after having read the value from the corresponding location(s), implying that it can now be written.

In the sliding windows buffer, the buffer administration stores the four pointers for the RW and WW [3]. This is the case when using the buffer implementation and communication library built in-house for Omphale. However, implementations supporting a single producer and consumer have no notion of the head pointers, r^{\wedge} and w^{\wedge} pointers. Their buffer administration would store only a *r* and a *w* pointer. The SoD buffers used in our target platform also do not store the head pointers in the buffer administration. Due to this issue, the sliding windows approach cannot be used directly.

The lack of the head pointers poses a second issue when acquiring locations in the sliding windows buffer. When an *acquire* statement is executed, it checks for the availability of space or data in the buffer from the current head pointer of WW or RW. However, the SoD buffers do not contain the head pointers. The SoD API provides functions namely *checkWrite* and *checkRead* which can check for the availability of space or data respectively from the current *w* and *r* pointers only. These functions have to be used in such a manner as to provide the ability to check for space or data from the head pointers of WW or RW. A mechanism is derived to account for the lack of these head pointers in the buffer administration while still being able to provide the required synchronization. This is explained in the following section.

Wrapper Generation

The synchronization statements use SoD API to implement their functionality. Hence a wrapper library has been built which makes use of the required SoD API. The lack of head pointers in the buffer administration is accounted for in the modified sliding windows buffer. The following listing gives a basic idea about the synchronization statements used in the modified sliding windows approach.

- ***AcquireSpace(n , $countSpace$)***
Check for ' $n + countSpace$ ' available spaces from w
If available, $countSpace$ is incremented by n
- ***ReleaseData(n , $countSpace$)***
Decrement $countSpace$ by n
Increment w by n
- ***AcquireData(n , $countData$)***
Check for ' $n + countData$ ' available data from r
If available, $countData$ is incremented by n
- ***ReleaseSpace(n , $countData$)***
Decrement $countData$ by n
Increment r by n

On executing *AcquireSpace*, it checks for the availability of $n + CountSpace$ spaces in the buffer. If spaces are available, it increments $countSpace$ by n . $countSpace$ is a counter which maintains the number of spaces that have been acquired in the buffer. On a subsequent *releaseData* call, it decrements $countSpace$. Similarly on executing *AcquireData*, it checks for the availability of $n + CountData$ data in the buffer. If data are available, it increments $countData$ by n . $countData$ is a counter which maintains the number of data that have been acquired in the buffer. On a subsequent *releaseSpace* call, it decrements $countSpace$.

The implementation of the synchronization wrapper library functions are shown in Figure 5.4a. Here, it can be seen that all the functions also take *port* as a parameter. The *port* refers to SoD ports that are part of SoD tasks. An SoD buffer is formed by the channel connecting two ports. *CheckRead* is a SoD API function that enables to check that there are n filled data locations from the current r pointer. Similarly, *CheckWrite* is a SoD API function that enables to check that there are n empty locations from the current w pointer. *UpdateWrite* and *UpdateRead* are SoD API functions that increment the w and r pointers respectively. It can be seen here that the *acquire* synchronization calls return control to the scheduler immediately, when there is not enough space or data available in the buffer.

A sample sequence of operations is shown in Figure 5.4 to illustrate the use of r and w pointers and the counters $countSpace$ and $countData$ when the synchronization calls are encountered. The value of the head pointers can be obtained by summing the corresponding tail pointers and counters as shown in Figure 5.4. The values that change on every step are shown in red. On executing *acqS* twice, a space is acquired and the values of cnt_S and the w^{\wedge} pointers are incremented by one each time. After a write is done, then an execution of *relD* increments the w pointer and cnt_S is decremented. In this case, the head of the write window has not moved and this is shown by w^{\wedge} retaining a value of 2.

5.2 Multiple Producers

Two tasks writing to the same buffer leads to the multiple producers problem. However, if the underlying communication library does not support a buffer with multiple producers, then two separate buffers need to be used. On using separate buffers, the consumer task does not know which of the two producers wrote to its corresponding buffer.

```

acquireDataSod(port, nrData, countData){
    if( OK != CheckRead(port, nrData + countData)
        return BLOCKED;
    else{
        if(countData + nrData >= buf_size)
            countData = (countData + nrData) % (buf_size)
        else
            countData = countData + nrData;
    }
}

releaseSpaceSod(port, nrData, countData){
    countData = countData + nrData;
    updateRead(port, nrData);
}

acquireSpaceSod(port, nrSpace, countSpace){
    if( OK != CheckWrite(port, nrSpace + countSpace)
        return BLOCKED;
    else{
        if(countSpace + nrSpace >= buf_size)
            countSpace = (countSpace + nrSpace) % (buf_size)
        else
            countSpace = countSpace + nrSpace;
    }
}

releaseDataSod(port, nrSpace, countSpace){
    countSpace = countSpace + nrSpace;
    updateRead(port, nrSpace);
}

```

(a) Wrapper Synchronization Library

Sequence of operations	W	cnt_S	r	cnt_D	Space/Data available	w^=w+cnt_S	r^=r+cnt_D
Initial	0	0	0	0	NA	0	0
acq_S(1, cnt_S)	0	1	0	0	Yes	1	0
acq_S(1, cnt_S)	0	2	0	0	Yes	2	0
Write 1 value							
relD(1, cnt_S)	1	1	0	0	NA	2	0

Figure 5.4: Example

5.2.1 Related Work

An approach using additional buffers [3] was described to solve the issue with multiple producers in an array. This shown in Figure 5.5. As can be seen, apart from the FIFO buffers s_{x0} and s_{x1} , two additional buffers $s_{x0'}$ and $s_{x1'}$ are added. The producer tasks write a value to these additional buffers indicating the indexes of the array which they have written to in the corresponding buffers. The consumer task checks the indexes in these additional buffers and compares it with the index that it needs to read, and reads the corresponding buffer. However, this approach cannot handle the case where there are multiple producers to the same array index.

Turjan et al. [22] propose a method to solve a similar issue as the previous one with

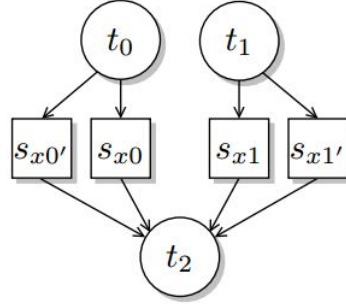


Figure 5.5: Addition of Additional buffers

multiple producers to an array. It is applicable only to NLPs with affine index-expressions. Affine expressions are a linear combination of loop parameters and a constant. By using affine index-expressions, exact data dependencies can be calculated and so the additional buffers are not needed. However, as in the previous approach, multiple producers to the same array index is not handled. Also, streaming applications contain non-manifest statements which cannot be handled by this approach.

In [23], Gangwal et al. propose a synchronization protocol which uses point-to-point FIFOs for communicating between tasks. However, they do not solve the multiple producers problem. Also in [24], a Task Transation Level (TTL) interface is provided to facilitate inter-task communication and synchronization. However, even in this case, the multiple producers problem is not solved.

Bijlsma et al. [3] solve the multiple producers problem by using a single buffer capable of having multiple producers. Multiple sliding windows are used per producer, and a consumer cannot read until all the multiple producers have indicated that the location which has multiple producers is not needed by all of them. However, in our case, the communication library supports only buffers with a single producer and consumer. Hence this approach is not applicable.

An approach that is directly applicable to solve the multiple producers problem could not be found.

5.2.2 Proposed Solution

To solve the multiple producers problem several approaches were proposed as discussed in section 4.1.2. However, these approaches address a related issue and not the exact problem that we would like to address. When separate buffers are used in the case of multiple producers, the problem is that the consumer task needs to identify which of the multiple producers wrote a value to its corresponding buffer. Due to single assignment requirement on the NLP it is known that these writes are mutually exclusive. Two approaches to solve the multiple producers problem are shown in the subsequent sections.

5.2.2.1 Data Valid Boolean

The consumer task has to identify which of the two producers wrote a value to its corresponding buffer. A data structure can be created encompassing a data valid boolean and the actual data to be passed in the buffer. This data structure is henceforth referred to as a container that is passed through the buffer. This is illustrated in Figure 5.6. Here the producer tasks are $P1$ and $P2$, and the consumer task is denoted as C . The producer task ($P1$) which actually writes a value sets the *data valid boolean* to *true* and writes the data into the container. The other producer task ($P2$) which does not write actual data to the container sets the *data valid boolean* to false. The consumer task extracts the *data valid boolean* from the containers in both the buffers. It reads the data from the buffer which has the *data valid boolean* set to true. In this approach, an empty data structure has to be passed through the buffers, corresponding to producers that do not need to write a value. If these buffers are located on the local memories of other processors, they could use up unwarranted bandwidth on the interconnect. The amount of bandwidth used depends on the size of the data that needs to be passed. Also, the overhead of the *data valid boolean* is smaller if bigger data structures need to be passed through the buffer as compared to smaller ones. This approach can be used for both shared memory and distributed memory systems.

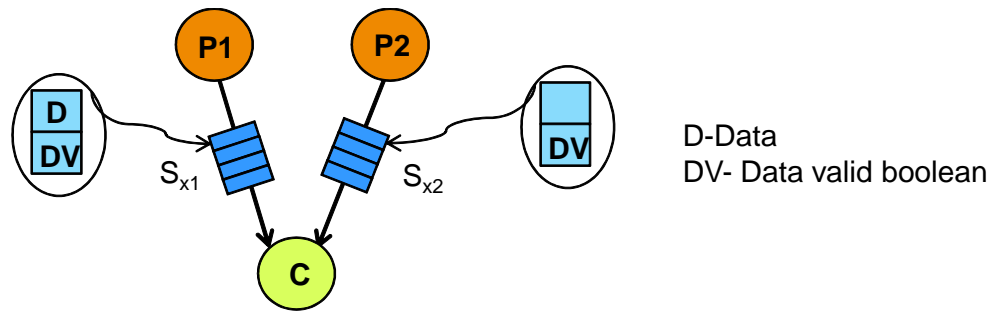


Figure 5.6: Multiple Producers - Data Valid Boolean

5.2.2.2 Valid / NULL Pointer

In order to identify which of the two producers wrote a value to its corresponding buffer, a valid/NULL pointer can be used. This is shown in Figure 5.7. The producer task which actually writes a value, passes a pointer to the data that has to be passed through the buffer. The data is stored in a shared memory which is accessible by the consumer task. The other producer tasks which do not need to write a value to the buffer, write a NULL value into the buffer. The consumer task reads the buffers corresponding to the different producer tasks, and checks which of them contain a valid pointer. Then the valid pointer is dereferenced to obtain the data stored in the shared memory. This approach can be used only for shared memory systems and not distributed memory systems. This is because, in distributed memory systems a pointer to a local memory location is not accessible remotely.

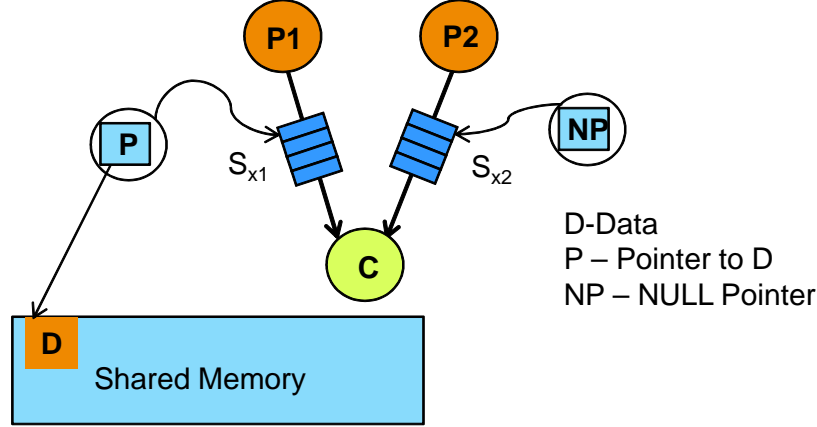


Figure 5.7: Multiple Producers - Valid/NULL Pointer

5.2.3 Implementation in Omphale

The proposed solution with NULL pointers has been implemented in Omphale. The existing implementation of Omphale is changed to obtain separate buffers for every producer. Figure 5.8 shows how a task is transformed so that multiple buffers can be generated per producer. The variable x is written twice, thereby making it a multiple producer. We create two new unique pointers x_uniq_p0 and x_uniq_p1 which replace the first and second occurrences of x where they are being written respectively. A conditional assignment statement is used in a *seq* block along with the statement that reads x . It assigns the value of x based on whether x_uniq_p0 or x_uniq_p1 has a valid address assigned to it. A *seq* block ensures that the statements in that block are made into a single task. As discussed earlier, it is known that only one of the two producers would write a valid pointer pointing to data value, into its corresponding buffer. The other producer would write a NULL pointer to the buffer. The function *isValid* checks for the valid pointer and assigns x by dereferencing the pointer.

<pre> loop{ if(~) x = f(); else x = g(); output(x); }while(1) </pre> <p style="text-align: center;">(a) Input Task</p>	<pre> loop{ if(~) *x_unq_p0 = f(); else *x_unq_p1 = g(); seq{ x = isValid(x_unq_p0) ? *x_unq_p0 : *x_unq_p1; output(x); } }while(1) </pre> <p style="text-align: center;">(b) Transformed Task</p>
--	--

Figure 5.8: Multiple Producers - Task Transformation

After parallelization, three tasks are obtained as seen in Figure 5.9. It can be observed that a NULL pointer is written into the buffer in `task_f` and `task_g` when a valid data value is not being written into the buffer.

<pre> do{ if (~) *x_unq_p0 = f (); else x_unq_p0 = NULL; } while (1) </pre>	<pre> do{ if (~) x_unq_p1 = NULL; else *x_unq_p1 = g (); } while (1) </pre>	<pre> do{ x = isValid(x_unq_p0) ? *x_unq_p0 : *x_unq_p1; output(x) } while (1) </pre>
(a) Task_f	(b) Task_g	(c) Task_Output

Figure 5.9: Parallel Tasks

5.3 Multiple Consumers

A buffer having multiple consumers leads to a multiple consumers problem. Separate buffers can be used for each consumer if the underlying communication library does not support buffers with multiple consumers. If one of the consumers does a conditional read, then the producer has to decide which of the two buffers should be written to and when it should be written.

5.3.1 Related Work

An approach using additional buffers [3] was described to solve a similar issue with multiple consumers in an array. This is shown in Figure 5.10. As can be seen, apart from the FIFO buffers s_{x0} and s_{x1} , two additional buffers $s_{x0'}$ and $s_{x1'}$ are added. The consumer tasks write a value to these additional buffers indicating the indexes of the array which they want to read from the producer. The producer task reads the additional buffers and writes a value to the corresponding buffer only when needed. In this case, the producer task has to perform additional synchronization and wait for the consumer task before being able to execute. Turjan et al. [22] address this issue only in the case of NLPs with affine-index expressions. In this approach, the additional buffers can be avoided. However, non-manifest statements are not supported.

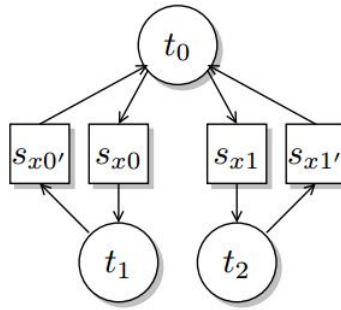


Figure 5.10: Addition of Additional buffers

As in the case of the multiple producers problem, in [23], Gangwal et al. propose a synchronization protocol which uses point-to-point FIFOs for communicating between tasks. However, they do not solve the multiple consumers problem. Also in [24], a Task Transaction Level (TTL) interface is provided to facilitate inter-task communication and synchronization. However, even in this case, the multiple consumers problem is not solved.

Bijlsma et al. [3] solve the multiple consumers problem by using a single buffer capable of having multiple consumers. Multiple read sliding windows are used corresponding to every consumer. Nevertheless, this approach is not directly applicable as our communication library supports only buffers with a single producer and consumer.

Few approaches for automatic parallelization do not solve the reordering problem or the issue with multiple producers and consumers. Sprint [25] and MAPS [26] perform automatic parallelization by taking C code as input. They use FIFO buffers for communication between tasks. However, they do not solve the above issues.

5.3.2 Proposed Solution

The approaches to solve the multiple consumers problem were presented in section 4.1.3. The demerits of these approaches were also discussed. To avoid the overhead of additional buffers, an approach is proposed whereby data is written to the buffers corresponding to all the consumers. The consumer which does not require the data due to a conditional read does not read the data, but updates the corresponding read pointer indicating that the particular location can now be used for writing. This can be referred to as unconditional synchronization, which has been proposed in [3].

5.3.3 Implementation in Omphale

<pre> loop{ x=f(); output1(x); if(~) output2(x); }while(1) </pre>	<pre> loop{ seq{ x=f(); x_unq_c0 = x; x_unq_c1 = x; } output1(x_unq_c0); if(~) output2(x_unq_c1); }while(1) </pre>
(a) Input Task	(b) Transformed Task

Figure 5.11: Multiple Consumers - Task Transformation

The problem of multiple consumers is solved with unconditional synchronization. The existing implementation of Omphale is changed to have two separate buffers, one per consumer. An input task is shown in Figure 5.11. Here the variable x is read by two functions *output1* and *output2*. It is to be noted that *output2* does a conditional read of x . In order to create separate buffers, we create two unique variable x_unq_c0 and x_unq_c1 . They are assigned to x and made part of a *seq* block. A *seq* block ensures that the statements in that block are contained in a single task, after parallelization. x_unq_c0 and x_unq_c1 also replace the first and second occurrences respectively, where x is being read. After parallelization, the three generated tasks are shown in Figure 5.12. It can be noted that an *else* branch is added in task_output2 where there is a conditional read of x . This branch contains synchronization statements that perform the unconditional synchronization.

Another alternative is to move the condition which leads to the conditional read, to the producer task such that the producer task now conditionally writes to the buffer corresponding to the consumer task that needs to read it. In this case, the producer

<pre> do{ x = f(); acqSpace(x_unq_c0); x_unq_c0 = x; relData(x_unq_c0); acqSpace(x_unq_c1); x_unq_c1 = x; relData(x_unq_c1); }while(1) </pre>	<pre> do{ acqData(x_unq_c0); output1(x_unq_c0); relSpace(x_unq_c0); }while(1) </pre>	<pre> do{ if(~) acqData(x_unq_c1); output2(x_unq_c1); relSpace(x_unq_c1); else acqData(x_unq_c1); relSpace(x_unq_c1); }while(1) </pre>
(a) Task_f	(b) Task_output1	(c) Task_output2

Figure 5.12: Parallel Tasks

task would not write to buffers corresponding to consumer tasks that do not require it, thereby saving unwanted writes. This is illustrated with an example shown in Figure 5.13. However, in this case, there is an increase in the number of buffers required. Here, two buffers are created for the variables c and d whereas only one is required in the previously proposed solution. The producer task now has to read the conditions from the appropriate buffers before writing to it. The tasks which write the conditions now have to write to two buffers as compared to a single one in the previously proposed solution. Due to these reasons, it is not preferred over the proposed solution.

<pre> loop{ c = g(); d = h(); x = f(); if(c) output1(x); if(d) output2(x); }while(1) </pre>	<pre> loop{ seq{ c = g(); c_unq_c0 = c; c_unq_c1 = c; } seq{ d = g(); d_unq_c0 = d; d_unq_c1 = d; } seq{ x = f(); if(c_unq_c0) x_unq_c0 = x; if(d_unq_c0) x_unq_c1 = x; } if(c_unq_c1) output1(x_unq_c0); if(d_unq_c1) output2(x_unq_c1); }while(1) </pre>
(a) Input NLP	(b) Transformed Task

Figure 5.13: Multiple Consumers - Alternate Solution

Non-Preemptive Scheduling

In non-preemptively scheduled systems, tasks yield the processor voluntarily. When blocking synchronization calls do not succeed in such tasks, they yield the processor voluntarily and return control back to the scheduler. Generated parallel tasks can contain multiple blocking synchronization statements to acquire space or data in the buffer. When a task blocks on executing one of them, control is returned back to the scheduler. However, the state of the task where it yielded control is not saved. When the task executes again, it starts from the beginning of the task. This can even lead to a deadlock as shown in section 4.2. In this chapter, a method for saving the state of the task, in order to prevent deadlock is proposed. A finite state machine (FSM) is generated wherein each state contains at most one blocking synchronization call.

Motivation

In high-performance embedded systems, non-preemptive scheduling is preferred over preemptive scheduling [27]. Non-preemptive scheduling algorithms are easier to implement than preemptive algorithms and have dramatically lower overhead at run-time. Non-preemptive scheduling is more efficient than preemptive scheduling since preemption incurs a context-switching overhead which can be significant in fine-grained multi-threaded systems [28]. Nonetheless, even in multiprocessor systems with preemptive processors, some co-processors or accelerators are non-preemptive. Some scheduling techniques also employ a combination of preemptive and non-preemptive scheduling approaches to provide efficient schedules [29]. It is therefore important that automatic parallelization tools are able to support non-preemptive multiprocessor systems.

Generated parallel tasks contain synchronization statements to prevent race conditions and deadlock. The *acquire* synchronization statement is blocking whereas the *release* synchronization statement is non-blocking. There are two *acquire* synchronization statements that are used in Omphale, namely *acquireSpace* and *acquireData*, which check for the availability of space or data respectively in the buffer. When space or data is not available in the buffer, the task blocks. On using a non-preemptive

scheduler, the task continues to wait until there is space or data available in the buffer. If the other task which releases space or data is scheduled on the same processor, it can lead to a deadlock. When a preemptive scheduler is used, the blocking task is preempted and a deadlock does not occur. In order that a task does not block indefinitely, causing a deadlock, a task yields control back to the scheduler voluntarily, when there is a lack of space or data in the buffer. This is helpful even on using preemptive schedulers as it might lead to better performance by preventing unnecessary waiting time for the task until it is preempted.

On using a preemptive scheduler, a higher priority task preempts a lower priority task. If the budget or time-slice within which a task can execute is exhausted, the task is preempted by the scheduler and the next task in the schedule is executed. In both cases, the state of the preempted task is saved implicitly by the scheduler. As a result, the task resumes from the statement at which it was preempted, on its next execution. When a preemptive scheduler is used, and a task gets blocked at a blocking synchronization statement, it yields control back to the scheduler and the state of the task is saved implicitly. On the next execution of the task, it resumes from the point at which it got blocked.

When a non-preemptive scheduler is used and a task blocks on a blocking synchronization statement, the task yields control back to the scheduler and the state of the task is not saved. When the same task is executed again, it resumes from the beginning of the task and not from the point where it was blocked. When the state of the task is not saved, it could lead to a deadlock as shown in section 4.2.

By generating an FSM inside the task, the state of the task can be saved. To the best of our knowledge, current automatic parallelization tools do not deal with the issues concerned with using non-preemptive schedulers along with blocking synchronization statements in tasks.

FSM Generation

In order to save the state of a task, an FSM is generated inside every task. From the input sequential description of a streaming application, Omphale generates parallel tasks. The extracted tasks generated by Omphale are part of a task graph and each task is organized as a tree containing the statements in a task. These statements include *while* loops, *for* loops, *if* statements, assignments, functions etc. The tree in every task is traversed recursively and a new tree of task statements is created which includes the state information. The states are generated such that there is at most one blocking synchronization call per state. The FSM is organized as *if* statement blocks containing statements which ought to execute, based on the value of a state variable *st*. The *if* blocks also contain a value for the next state to which the FSM should transition. A template of a generated FSM is shown in Figure 6.1. The FSM is organized in this manner so that if blocking synchronization calls succeed, they can proceed to the next state. If these calls fail, the task returns to the same state where it was blocked earlier. Once a task finishes its execution, it should return control back to the scheduler. Therefore, the last state in the FSM contains a return statement.

The state values are constants which are represented starting with the letter *c* in the task templates with states. These are computed at compile time in Omphale by keeping track of the number of states as the new tree is generated for the FSM. The states generated for every *statement* are denoted by *States_Statement* as shown in Figure 6.2. Depending

```

    if(st == ~){
        ..
        st = ~;
    }
    .
    if(st == ~){
        ..
        st = ~;
    }

```

Figure 6.1: FSM template

on the type of *statement*, there could be one or more states generated for a statement. The states generated for a statement are captured in a tree containing *if* blocks. The *if* statements check for the value of the *st* variable and do an update of the *st* variable corresponding to the next state.

```

    if(st == c_1){
        ..
        st = ~;
    }
    .
    if(st == c_n){
        ..
        st = ~;
    }

```

Figure 6.2: *States_Statement*

The approach for generating states for different statements is described in the following sections.

6.1 Infinite While Loop

An infinite while loop corresponds to a *do{} while(1)* loop. In a streaming application, the infinite while loop is usually the outermost loop. Especially in the case of the SoD kernel, the scheduler calls each of the tasks to be run on a processor in a round-robin fashion. Hence this outermost while loop can be removed. However, after finishing the execution of a task, control should be returned to the scheduler by means of a *return* statement. Thereby, the last state in the case of an infinite while loop has a *return* statement at the end. Also the next state is assigned as the first state in the while loop. The statements inside the while loop can be either if-else, assignment statements, function statements or for loops. The generation of states for these statements is explained in the corresponding sections.

A template of a task, with and without states is shown in Figure 6.3. This highlights how a task without states is transformed into a task with an FSM. It can be seen from Figure 6.3b that the last state has a *return* statement and an assignment of the *next* state as the first state in the loop. Here, *st* is a variable which keeps track of the current state in the FSM. Let N_{wb} be the number of states in the body of the infinite while loop. N_{wb} corresponds to the number of *if* blocks present in *States_Statements* shown in Figure 6.3b. The number of states corresponding to an infinite while loop statement

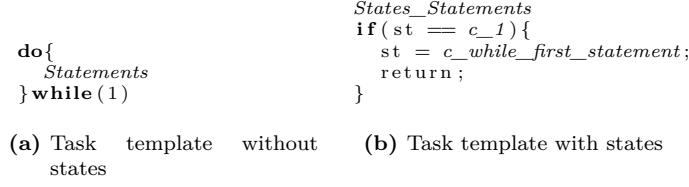


Figure 6.3: While Loop Template

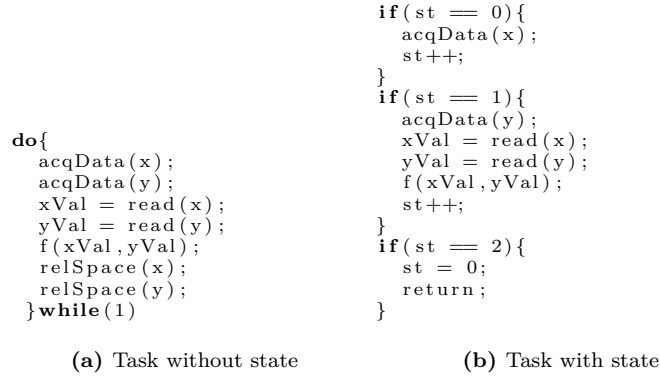


Figure 6.4: While loop Example

N_w can be calculated from Figure 6.3b and is given in Equation 6.1.

$$N_w = N_{wb} + 1 \quad (6.1)$$

An example of a task with a while loop and a function statement is shown in Figure 6.4. Here the function f reads needs two values x and y which it reads from corresponding buffers. There are two blocking *acquire* statements which leads to two different states in the FSM. The last state in the FSM contains a *return* statement and an assignment of the *next* state as the first state in the FSM.

6.2 Conditional While Loop

A conditional while loop corresponds to a *do{} while(condition)* loop. The loop is executed as long as the condition is satisfied. A template of a task with a conditional while loop, with and without states is shown in Figure 6.5. In the last state of the while loop, if the condition is satisfied, the next state is assigned as the first state in the loop. Otherwise, the next state is assigned as the state after the while loop by a *st++*. An example of a conditional while loop with states is shown in Figure 6.6. Let N_{wcb} be the number of states in the body of a conditional while loop. N_{wcb} corresponds to the number of *if* blocks present in *States_Statements* shown in Figure 6.5. The number of states corresponding to a conditional while loop statement N_{wc} is given in Equation 6.2.

$$N_{wc} = N_{wcb} + 1 \quad (6.2)$$

<pre> do{ <i>Statements</i> }while (cond) </pre>	<pre> <i>States_Statements</i> if (st == c_1){ if (cond){ st = c_while_cond_first_statement; return; } else{ st++; } } </pre>
(a) Task template without states	(b) Task template with states

Figure 6.5: Conditional While Loop Template

<pre> do do{ acqData(x); acqData(y); xVal = read(x); yVal = read(y); f(xVal,yVal); relSpace(x); relSpace(y); acqData(whileCond); whileCondVal = read(whileCond); relSpace(whileCond); }while(whileCondVal); }while(1); </pre>	<pre> if (st == 0){ acqData(x); st++; } if (st == 1){ acqData(y); xVal = read(x); yVal = read(y); f(xVal,yVal); relSpace(x); relSpace(y); st++; } if (st == 2){ acqData(whileCond); whileCondVal = read(whileCond); relSpace(whileCond); st++; } if (st == 3){ if (whileCondVal){ st = 0; return; } else{ st++; } } if (st == 4){ st = 0; return; } </pre>
(a) Task without state	(b) Task with state

Figure 6.6: Conditional While loop Example

6.3 Nested While Loop

A nested while loop can comprise of an conditional while loop nested inside an infinite while loop. Alternatively, conditional while loops can be nested inside each other. An example of FSM generation for a nested while loop is shown in Figure 6.7.


```

do{
  do{
    do{
      acqData(x);
      acqData(y);
      xVal = read(x);
      yVal = read(y);
      f(xVal,yVal);
      relSpace(x);
      relSpace(y);
      acqData(whileCondn_1);
      whileCondnVal_1 = read(whileCondn_1);
      relSpace(whileCondn_1);
    }while(whileCondn_1);
    acqData(whileCondn_2);
    whileCondnVal_2 = read(whileCondn_2);
    relSpace(whileCondn_2);
  }while(whileCondn_2);
}while(1);

```

```

if(st == 0){
  acqData(x);
  st++;
}
if(st == 1){
  acqData(y);
  xVal = read(x);
  yVal = read(y);
  f(xVal,yVal);
  relSpace(x);
  relSpace(y);
  st++;
}
if(st == 2){
  acqData(whileCondn_1);
  whileCondnVal_1 = read(whileCondn_1);
  relSpace(whileCondn_1);
  st++;
}
if(st == 3){
  if(whileCondnVal_1){
    st = 0;
    return;
  }
  else{
    st++;
  }
}
if(st == 4){
  acqData(whileCondn_2);
  whileCondnVal_2 = read(whileCondn_2);
  relSpace(whileCondn_2);
  st++;
}
if(st == 5){
  if(whileCondnVal_2){
    st = 0;
    return;
  }
  else{
    st++;
  }
}
if(st == 6){
  st = 0;
  return;
}

```

(a) Task without state

(b) Task with state

Figure 6.7: Nested While Loop Example

6.4 For Loop

The *for* loop supported in OIL is of the following type - *forloop(lower_iterator <= var <= upper_iterator)*. This is translated to a for loop in C, of the following type *for(var=lower_iterator; var <= upper_iterator; var++)*. For loops can be used to access the different indices of an array. The template of a *for* loop with and without states is shown in Figure 6.8. A *for* loop can be represented with a single state, as can be seen in Figure 6.8b. An *if* statement is generated in this state to ensure that the statements inside the loop are executed as long as the upper and lower boundary conditions in the for loop are satisfied. The *index* variable is declared as *static* and initialized with the lower bound of the *for* loop. If a task blocks inside a *for* loop due to lack of space or data, the control is returned back to the scheduler. When the task executes again, it jumps to the state corresponding to the

for loop and resumes execution based on the value of the *index* variable. An example of the generation of an FSM for a *for* loop is shown in Figure 6.9.

<pre> for(index = lb; index < ub; index++){ Statements; } </pre>	<pre> if(st == c_1){ if(index >= lb && index < ub){ Statements; index = index + 1; } else{ st++; } } </pre>
(a) Task template without states	(b) Task template with states

Figure 6.8: For Loop Template

<pre> for(i = 0; i < 4; i++){ acqData(x); read(x, i); relSpace(x); } </pre>	<pre> if(st == 0){ if(i >= 0 && i < 4){ acqData(x); read(x, i); relSpace(x); i = i + 1; } else{ st++; } } </pre>
(a) Task without state	(b) Task with state

Figure 6.9: For Loop Example

6.5 Function Statement

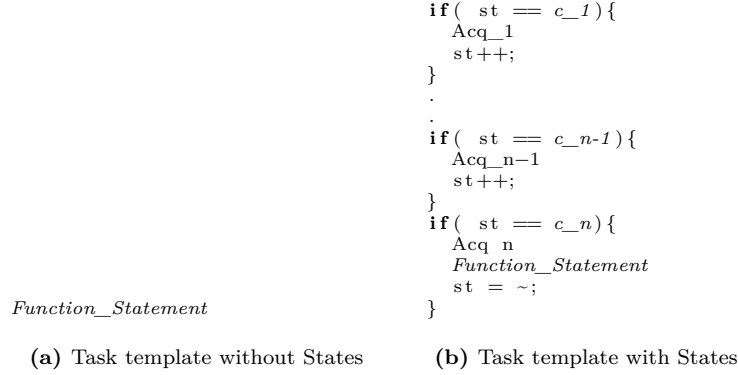
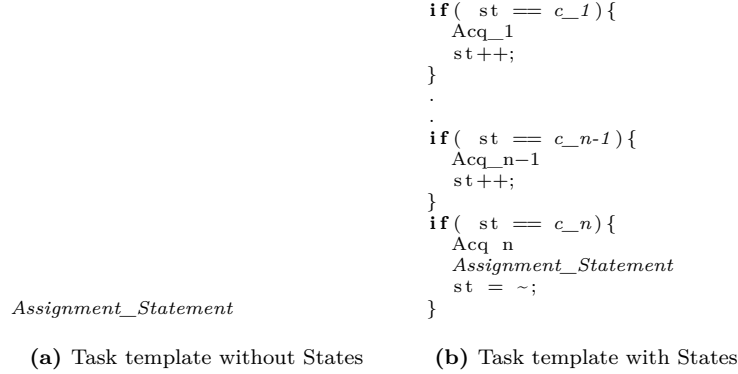
A function statement can contain more than one blocking *acquire* statement. A state is generated per blocking *acquire* statement. The template for a function statement having *n* *acquires* is shown in Figure 6.10. The number of states generated for a function statement is equal to the number of blocking *acquire* statements.

6.6 Assignment Statement

An assignment statement can also contain more than one blocking *acquire* statement. As in the case of a function statement, a state is generated per blocking *acquire* statement. The template for an assignment statement having *n* *acquires* is shown in Figure 6.11. The number of states generated for an assignment statement is equal to the number of blocking *acquire* statements.

6.7 If-Else Statement

The template of a task with an *if – else* statement is shown in Figure 6.12a. Before checking the condition of an *if* statement, the value of the condition is read from

**Figure 6.10:** Function Statement**Figure 6.11:** Assignment Statement

corresponding buffer(s). These statements are denoted as *Read_cond_Statements*. The corresponding states generated are referred to as *States_Read_cond_Statements*. The following state in the FSM is used to decide the next state to which the FSM should transition based on the satisfiability of the condition. If the condition is satisfied, it transitions to the following state, which corresponds to the *statements* in the *if* body. In case the condition is not satisfied, the FSM transitions to the first state corresponding to the statements in the *else* branch. After the last state corresponding to the statements in the *if* branch, the FSM transitions to the state after the statements in the *else* branch. An example of the generation of an FSM for an if-else statement is shown in Figure 6.13.

Let N_{ib} be the number of states corresponding to *States_If_body_Statements* in Figure 6.12b. Let N_{eb} be the number of states corresponding to *States_Else_body_Statements* in Figure 6.12b. Let N_{rc} be the number of states corresponding to *States_Read_cond_Statements* in Figure 6.12b. Then the number of states corresponding to a *if-else* statement, N_{ie} is given in Equation 6.3.

$$N_{ie} = N_{ib} + N_{eb} + N_{rc} + 2 \quad (6.3)$$

	<pre> States_Read_cond_Statements if(st == c_1){ if(cond) st++; else st = c_1 + N_ib + 2; } States_If_Body_Statements if(st == c_2{ st = c_2 + N_eb + 1 ; } States_Else_Body_Statements </pre>
<pre> Read_cond_Statements if(cond){ If_Body_Statements } else { If_Body_Statements } </pre>	
(a) Task template without states	(b) Task template with states

Figure 6.12: If-Else Statement

	<pre> if(st == 0){ acqData(x); cond = read(x); relSpace(x); st++; } if(st == 1){ if (cond) st++; else st = 4; } if(st == 2){ acqSpace(y); write(y,g()); relData(y); st++; } if(st == 3){ st = 5; } if(st == 4){ acqSpace(y); relData(y); st++; } if(st == 5){ st = 0; return; } </pre>
<pre> do{ acqData(x); cond = read(x); relSpace(x); if(cond){ acqSpace(y); write(y,g()); relData(y); } else { acqSpace(y); relData(y); } } while(1) </pre>	
(a) Task without states	(b) Task with state

Figure 6.13: If-Else Example

6.8 Nested If-Else Statements

In the case of nested if-else statements, it should be ensured that transitions in the FSM are made to the correct statements for each of the corresponding nested if statements. An example of nested *if* statements is shown in Figure 6.14. It can be seen that with a level of nesting of two (if inside if) and one blocking *acquire* per *if* branch, the number of states is 11.

```

do{
    acqData(condBuf_1);
    cond_1 = read(condBuf_1);
    relSpace(condBuf_1);
    if(cond_1){
        acqData(condBuf_2);
        cond_2 = read(condBuf_2);
        relSpace(condBuf_2);
        if(cond_2){
            acqSpace(y);
            write(y,g());
            relData(y);
        }
        else{
            acqSpace(y);
            relData(y);
        }
    }
    else {
        acqData(condBuf_2);
        relSpace(condBuf_2);
        acqSpace(y);
        relData(y);
    }
}while(1)

```

```

if(st == 0){
    acqData(condBuf_1);
    cond_1 = read(condBuf_1);
    relSpace(condBuf_1);
    st++;
}
if(st == 1){
    if (cond_1) st++;
    else st = 8;
}
if(st == 2){
    acqData(condBuf_2);
    cond_2 = read(condBuf_2);
    relSpace(condBuf_2);
    st++;
}
if(st == 3){
    if (cond_2) st++;
    else st = 6;
}
if(st == 4){
    acqSpace(y);
    write(y,g());
    relData(y);
    st++;
}
if(st == 5){
    st = 7;
}
if(st == 6){
    acqSpace(y);
    relData(y);
    st++;
}
if(st == 7){
    st = 10;
}
if(st == 8){
    acqData(condBuf_2);
    relSpace(condBuf_2);
    st++;
}
if(st == 9){
    acqSpace(y);
    relData(y);
    st++;
}
if(st == 10){
    st = 0;
    return;
}

```

(a) Task without states

(b) Task with state

Figure 6.14: Nested If-Else Example

Saving Variables in Task-State

Once an FSM is generated inside a task, there would be variables that are accessed across states. It has to be ensured that the values written to these variables are not lost, when control is returned to the scheduler on encountering a blocking synchronization statement. This is done by saving them to the task-state, which is maintained by the SoD kernel for every task. The values of variables stored in the task-state are held for the life-time of the application, like *static* variables. The state variable *st* and the counter variables used in the modified sliding windows buffer are saved in the task-state. When there are variables with multiple readers, the producer task contains different writes of the produced value to each of the multiple consumers. Therefore, this produced value has to be maintained across states and is thereby stored in the task-state.

Optimizations

In this chapter, different optimizations to improve the performance or reduce the static memory requirements of the generated tasks are discussed. Section 7.1 presents how the states in the FSMs generated in tasks are reduced. Section 7.2 presents how the task-state can be reduced. Section 7.3 discusses how the synchronization overhead can be reduced. In the remainder of the thesis, *acquire* statements are also referred to as *acquires* and *release* statements as *releases*. These statements constitute the synchronization statements in the task. All the other statements are referred to as non-synchronization statements (*NSS*).

7.1 Reduction of States in FSM of tasks

The number of states in a task depend mainly on the number of blocking (*acquire*) synchronization statements. If the number of states can be reduced, then the corresponding check for the state and the update of the state variable can be removed. This could result in an increase in throughput.

7.1.1 Moving Acquire/Release outside if-else statements

The number of states can be reduced by removing a pair of *acquire/release* statements from an *if* block and moving another pair out of the corresponding *else* block or vice-versa. In order to perform unconditional synchronization on a buffer, *acquire* and *release* statements are also placed in the *if* block or the *else* block where data is not read or written to that buffer. Therefore there are two pairs of *acquire, release* statements, one in the *if* block, and the other in the *else* block. If these *if – else* blocks do not contain loops, then one pair of *acquire/release* statements can be removed. Then the *acquire* of the other pair can be moved just before the *if – else* block and the corresponding *release*, can be moved after the *if – else* block. This reduces the number of *acquires* by a factor of 2, thereby reducing the number of states. It is important to note that we still perform unconditional synchronization after making the above changes.

<pre> do{ acqData(x); cond = read(x); relSpace(x); if(cond){ acqSpace(y); write(y,g()); relData(y); } else { acqSpace(y); relData(y); } }while(1) </pre>	<pre> if(st == 0){ acqData(x); cond = read(x); relSpace(x); st++; } if(st == 1){ if (cond) st++; else st = 4; } if(st == 2){ acqSpace(y); write(y,g()); relData(y); st++; } if(st == 3){ st = 5; } if(st == 4){ acqSpace(y); relData(y); st++; } if(st == 5){ st = 0; return; } </pre>
(a) Task without states	(b) Task with states

Figure 7.1: Before moving acquires out of if-else statements

<pre> do{ acqData(x); cond = read(x); relSpace(x); acqSpace(y); if(cond){ write(y,g()); } relData(y); }while(1) </pre>	<pre> if(st == 0){ acqData(x); cond = read(x); relSpace(x); st++; } if(st == 1){ acqSpace(y); if(cond){ write(y,g()); } relData(y); st++; } if(st == 2){ st = 0; return; } </pre>
(a) Task without states	(b) Task with states

Figure 7.2: After moving acquires out of if-else statements

The reduction in states by the above mentioned approach is illustrated by an example. Figure 7.1 shows the generation of states before moving *acquires* out of an *if – else* block. It can be seen that there are two pairs of *acqSpace(y)/relData(y)* statements inside the *if – else* block in Figure 7.1a. The corresponding FSM in Figure 7.1b has six states. Figure 7.2 shows the generation of states after moving acquires out of the *if – else* block. It can be observed that out of the two pairs of *acqSpace(y)/relData(y)* statements in Figure 7.1a, one has been removed and the other has been moved outside the *if – else* block. Since there are no other statements inside the *else* block, it has been removed. The number of states in the corresponding FSM, seen in Figure 7.2b is

three. Therefore the number of states has been reduced by three. However, it can be seen that the variable *cond* is used across states containing blocking *acquire* statements and therefore has to be saved in the task-state.

7.1.2 Combining Acquires

The number of states in a FSM in a task mainly depends on the number of *acquire* statements in the task. By combining *acquires*, the number of states can be reduced. This is illustrated in Figure 7.3 where n *acquires* are combined together. The *acquire* function is split into two functions, one performing the check for space or data, which is called *acquireCheck* and the other which performs an update of the corresponding *countSpace* and *countData* counters, which we call *Update*. Therefore the *AcquireData* function is split into *AcquireDataCheck* and *UpdateData*. Similarly, the the *AcquireSpace* function is split into *AcquireSpaceCheck* and *UpdateSpace*. In Figure 7.3, *acquireDataCheck* and *AcquireSpaceCheck* are represented by *AcqC*. The checking of space or data accordingly is done for all the *acquires* and only if all of them succeed, it proceeds to updating the count variables present in the corresponding *Upd* functions. This is done by means of the *and* operator. Even if one of the *acquire* calls gets blocked due to lack of space or data, control is returned back to the scheduler. Only if all the calls succeed, the corresponding *countSpace* or *countData* counters are incremented.

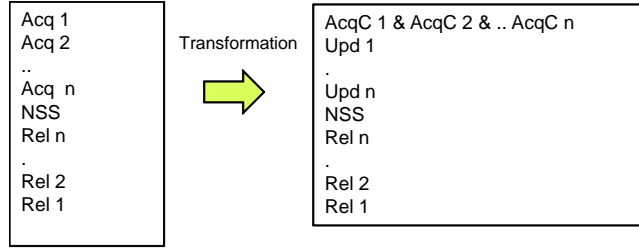


Figure 7.3: Combining Acquires

The code transformations as seen in Figure 7.3 involve moving *acquires* above *NSS* and *releases*. According to Theorem 7.1 and Theorem 7.2 these transformations preserve deadlock freedom, provided the conditions mentioned in the theorems are satisfied. These theorems are detailed in the subsequent sections.

Combining acquires can result in a decrease in throughput. For example, consider the task in Figure 7.3 with n acquires combined together. Consider the case that a task can get blocked on the last *acquire* and returns control back to the scheduler. On the next execution of the task, the $n - 1$ *acquires* have to be executed again which may lead to a decrease in throughput. Also combining *acquires* requires that they are first moved over *NSS* or *releases* which may also decrease the throughput. The reason for the probable decrease in throughput is mentioned in the following sections on moving *acquires*.

In order to combine acquires, certain code transformations need to be applied so that they do not cause deadlock. These code transformations include moving *acquires* above *NSS* or *releases*, to be able to combine them. The conditions under which these code transformations can be applied are analyzed in section 7.1.2.1 and 7.1.2.2.

7.1.2.1 Moving Acquires - Type I

In order to combine *acquire* statements, they have to be moved over other statements, so that they can be executed one after another. Type I refers to *acquire* statements that can be moved above NSS. A task template containing acquires is shown before and after transformation in Figure 7.4. The acquire statements and the corresponding release statements in the task are numbered from 1 to n as seen in Figure 7.4. The NSS are also numbered accordingly from 1 to n . The transformation step involves moving the acquires above the NSS. After transformation, it can be noted from Figure 7.4, that the $n-1$ *acquires* have been moved above the other NSS so that the *acquires* can be executed one after another. It is important to note here that none of the *acquires* have been moved over a *release* statement.

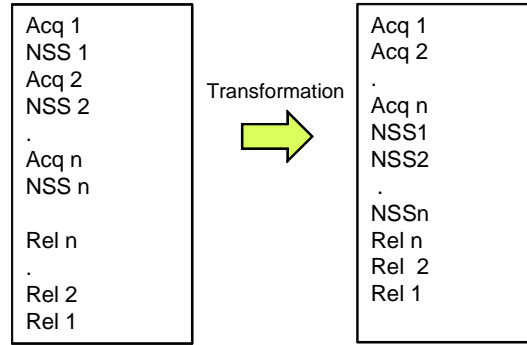


Figure 7.4: Task Transformation - Moving Acquires - Type I

Deadlock Freedom

We should prove that moving *acquires* above NSS, does not cause a deadlock. Here we prove deadlock freedom by creating an HSDF model of the tasks before and after the transformation. The tasks contain a sequence of *Acquires*, *Releases* and *NSS*. Conditional statements such as *if – else* and conditional while loops having *acquire* and *releases* in them cannot be represented using an HSDF model.

Construction of HSDF Model

A task graph contains various tasks that communicate via circular buffers. The tasks contain *acquire* and *release* synchronization statements and NSS. An HSDF model is created such that every *acquire* statement is represented by an acquire actor. Similarly every *release* statement is represented by a *release* actor and every NSS by an NSS actor. From a task graph TG , an HSDF model G is created as follows. For every task T_n present in TG , an HSDF model G_n is created containing *acquire*, *release* and *NSS* actors which belong to T_n . These actors are connected by edges representing the sequential dependencies between them. An edge is created from the last actor in G_n to the first actor with an initial token. Thereby a cycle is created with an initial token. This cycle ensures that the next execution of the task can happen only after finishing the current execution.

After all the tasks in TG have been represented in G , a directed edge is created from every *releaseSpace* actor to its corresponding *acquireSpace* actor. Similarly, a directed

edge is created from every *releaseData* actor to its corresponding *acquireData* actor. By doing so, cycles are formed across sub-graphs. These are called external cycles. To prevent deadlock, initial tokens are added on every cycle. The initial token is not placed on any of the edges inside G_n between *acquire*, *release* and NSS actors. It is placed on edges that are outside G_n .

Figure 7.5 shows the HSDF model of the tasks shown in Figure 7.4 with two *acquires* and two *releases*. We have actors corresponding to *acquires*, *releases* and NSS in a task. The edges denote dependencies between execution of actors. Each actor has a positive execution time. An actor fires when it has input tokens on all its incoming edges. The sequential execution of the statements in a task is guaranteed by the edges between their corresponding actors. It can be seen from Figure 7.5 that the actors are ordered in the same manner as the statements in the corresponding task. The *acquire* actors have incoming edges from corresponding *release* actors (not shown) of the corresponding buffer. Similarly, *release* actors have outgoing edges to *acquire* actors (not shown) of the corresponding buffer. It can be seen from Figure 7.5 that after transformation, the actor *Acq 2* has been moved above *NSS1* and *NSS2*. We will now introduce some definitions which would be used in the proof for deadlock freedom.

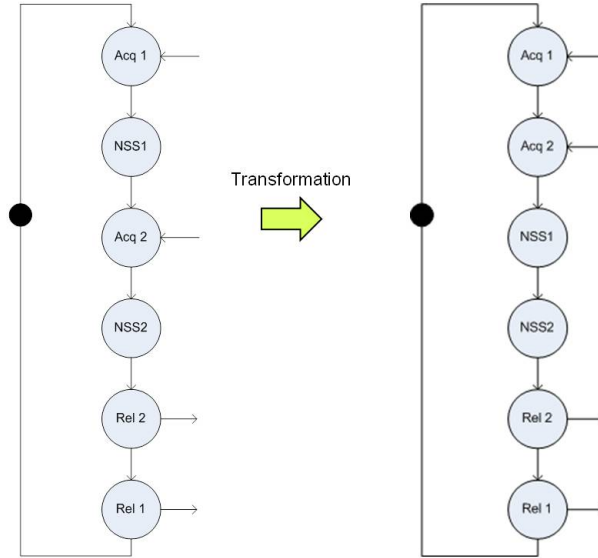


Figure 7.5: Moving Acquires - Type I - Dataflow Model

Definitions

Definition 7.1 (Actor): Assume P is a set of *Ports*. An actor a is a tuple (I, O, t, id) consisting of $I \subseteq P$ input ports denoted by $I(a)$ and set of $O \subseteq P$ output ports denoted by $O(a)$ with $I \cap O = \phi$. Assume A is a set of actors. The type of an actor is given by the function $t : A \rightarrow Type$ where $Type = \{Acq, Rel, NSS\}$. A set of *acquire* actors is denoted by $AcqS = \{a \in A | t(a) = Acq\}$ and a set of *release* actors is denoted by $RelS = \{a \in A | t(a) = Rel\}$. Every actor has a task-id given by the function $id : A \rightarrow \mathbb{N}$.

Definition 7.2 (HSDF Graph): An HSDF graph G is a tuple (A, D) consisting of a finite set A of actors and a finite set $D \subseteq Ports \times Ports$ of dependency edges. The

source of a dependency edge is an output port of some actor, the destination is an input port of some actor. The operator $SrcA$ gives the source actor of a dependency edge and correspondingly $DstA$ gives the destination actor of a dependency edge. The operator $SrcP$ gives the source port of a dependency edge and similarly $DstP$ gives the destination port of a dependency edge. All ports of all actors are connected to precisely one edge, and all edges are connected to ports of some actor. For every actor $a = (I, O, t, id) \in A$, we denote the set of all dependency edges that are connected to the ports in I by $InD(a)$. Similarly the set of all dependency edges connected to the ports in O are denoted by $OutD(a)$.

Definition 7.3 A task T_n is modeled by an HSDF graph $G_n = (A_n, D_n)$ which is a sub-graph of the HSDF graph $G = (A, D)$ which models the complete task graph. G_n is a sub-graph of G with $A_n = \{a \in A \mid id(a) = n\}$, $AcqS_n = \{a \in AcqS \mid id(a) = n\}$, $RelS_n = \{a \in RelS \mid id(a) = n\}$

Definition 7.4 (*Acquire-Release path*) : An *acquire – release* path is defined as a simple directed path within a sub-graph G_n from an *acquire* actor to a *release* actor containing actors belonging to G_n . It is denoted as $(cpsrc, \dots, cpdst)$ where $cpsrc \in AcqS_n$ and $cpdst \in RelS_n$. No initial tokens are present on this path by construction. There exists at least one dependency edge $(a, cpsrc) \in D$ such that $InD(cpsrc) = (a, cpsrc)$ where $id(a) \neq n$. This refers to the incoming edge of an *acquire* actor from an actor outside G_n . Similarly, there exists at least one dependency edge $(cpdst, a) \in D$ such that $OutD(cpdst) = (cpdst, a)$ where $id(a) \neq n$. This refers to the outgoing edge of a *release* actor to an actor outside G_n .

For example in Figure 7.6, G_1 has four *acquire – release* paths namely $(Acq1, Rel1)$, $(Acq1, Rel2)$, $(Acq2, Rel1)$, $(Acq2, Rel2)$. It can also be noted that there are no initial tokens on any of these *acquire – release* paths.

Definition 7.5 : An external cycle is formed by two paths, namely an *acquire – release* path $(cpsrc, \dots, cpdst)$ belonging to G_n , and an external path from $cpdst$ to $cpsrc$ containing actors belonging to G_m where $m \neq n$. Such an external cycle can be observed in Figure 7.6, consisting of the *acquire – release* path $(Acq1, Rel2)$ within G_1 and a path through G_2 from $Rel2$ to $Acq1$.

Lemma 7.1 When *acquire* actors are moved above *NSS* actors in a sub-graph G_n of G to obtain G_n' , the number of initial tokens on every corresponding external cycle remains the same.

PROOF : Assume G is deadlock-free. This implies that all cycles in G have initial tokens. In the sub-graph G_n , there exists an *acquire – release* path from every *acquire* actor to every *release* actor (by definition 7.4). No initial tokens are present on this path by construction, as mentioned in definition 7.4. Therefore, all initial tokens on external cycles formed by *acquire – release* paths in G_n are present on paths outside G_n . This implies that this transformation does not change the number of initial tokens present on external cycles. ■

Lemma 7.2 : When *acquire* actors are moved above *NSS* actors in a sub-graph G_n of G , G_n' is obtained wherein cardinality of the set of *acquire – release* paths remains the same.

PROOF : Given a set of *acquire – release* paths CP which belong to G_n . We have to prove that after *acquire* actors are moved above *NSS* actors, $|CP'| = |CP|$. A

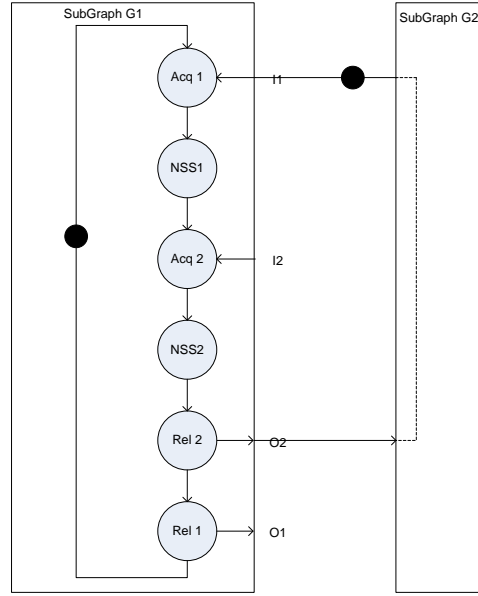


Figure 7.6: Illustration of External cycle

NSS actor has no dependency edges from any actor outside G_n by definition. By definition 7.4, an acquire-release path is denoted by $(cpsrc, cpdst)$ where $cpsrc \in AcqS$ and $cpdst \in RelS$. Therefore, though the actual path between $cpsrc$ and $cpdst$ might change upon moving *acquire* actors above *NSS* actors, the number of *acquire* – *release* paths do not change. ■

Theorem 7.1 *An HSDF model G of a task graph TG , transformed to G' by moving acquire actors above *NSS* actors, is deadlock-free, provided G is deadlock-free and none of the acquire actors is moved over a release actor.*

PROOF : Given a deadlock-free graph G implies that the all the cycles present in this graph have initial tokens. To prove deadlock freedom after moving acquires above *NSS*, we should prove that all cycles in G' have initial tokens.

We consider a sub-graph G'_n of G' corresponding to a task T'_n within which *acquire* actors are moved above *NSS* actors. The rest of the graph G' remains intact. The number of possible external cycles is equal to the number of *acquire* – *release* paths in G_n . By Lemma 7.2, the number of *acquire* – *release* paths remain the same. This implies that no additional external cycle is added in G'_n because external cycles are formed by *acquire* – *release* paths. By Lemma 7.1 the number of initial tokens present on external cycles remain the same. Therefore all external cycles of G'_n contain initial tokens and thereby the transformation in G_n does not cause a deadlock.

Similarly, for every sub-graph G'_n the transformation does not cause a deadlock. Therefore all cycles in G' have initial tokens and G' is deadlock-free after this transformation. ■

An illustration of the proof can be seen in Figure 7.7. It shows the HSDF models of a task with two *acquires* and two *releases*, before and after an *acquire* has been moved over *NSS*. The initial tokens are not present on cycle paths $(Acq1, Rel1)$, $(Acq1, Rel2)$, $(Acq2, Rel1)$, $(Acq2, Rel2)$ and hence they remain intact on

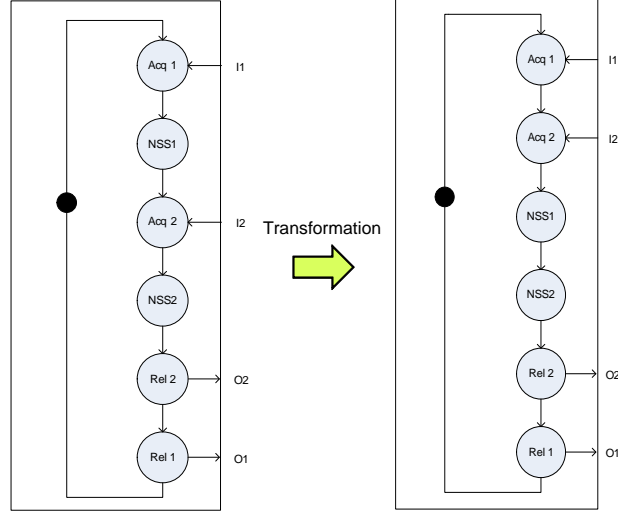


Figure 7.7: Illustration of Proof

the external cycles after transformation. It can also be seen that when *Acq2* is moved over *NSS1*, some *acquire – release* paths have different paths between their source and destination actors. For example, (*Acq2*, *Rel2*) has a new actor *NSS1* in its path. However the number of *acquire – release* paths remains the same. Since the number of possible external cycles corresponds to the number of *acquire – release* paths and this has not changes, no additional external cycle has been added. Therefore all the external cycles contain initial tokens and hence the transformation does not introduce a deadlock.

7.1.2.2 Moving Acquires - Type II

Type II refers to *acquire* statements that can be moved above other statements including *release* statements. A task template containing *acquires* is shown before and after transformation in Figure 7.8. The transformation step involves moving the *acquires* above the *NSS* and *release* statements. After transformation, it can be noted from Figure 7.8, that *Acq2* has been moved above *NSSa* and *NSSb* so that the *Acq1* and *Acq2* can be executed one after another.

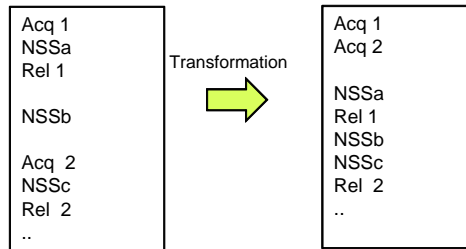


Figure 7.8: Task Transformation

Deadlock Freedom

We prove deadlock freedom by creating an HSDF model of the tasks before and after transformation. Every SDF graph can be converted into its equivalent HSDF graph as described in [6]. Figure 7.9 shows the HSDF model of the tasks shown in Figure 7.8 with two *acquires* and two *releases*. We have actors corresponding to *acquires*, *releases* and NSS in a task. It can be seen from Figure 7.9 that after transformation, the actor *Acq 2* has been moved above *NSS1* and *NSS2* and *Rel1*.

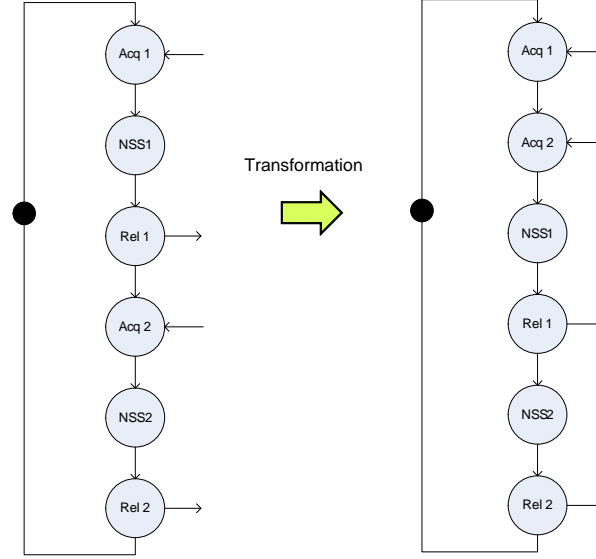


Figure 7.9: HSDF Model

Lemma 7.3 When *acquire* actors are moved above *release* actors in a sub-graph G_n of G to obtain G_n' , the initial tokens present on external cycles remain intact.

PROOF : It is given that G is deadlock-free. This implies that all cycles in G have initial tokens. In the sub-graph G_n , there exists a *acquire* – *release* path from every *acquire* actor to every *release* actor (by definition 7.4). No initial tokens are present on an *acquire* – *release* path by construction, as mentioned in definition 7.4. Therefore, all initial tokens on external cycles formed by *acquire* – *release* paths in G_n are present on paths outside G_n . This implies that this transformation does not alter the initial tokens present on external cycles. ■

Lemma 7.4 : When an *acquire* actor is moved above a *release* actor in a sub-graph G_n of G , G_n' is obtained wherein the number of *acquire* – *release* paths is increased.

PROOF : Given a set of *acquire* – *release* paths CP which belong to G_n . We have to prove that after an *acquire* actor is moved above a *release* actors, *acquire* – *release* paths $|CP'| \neq |CP|$. By definition 7.4, a *acquire* – *release* path is denoted by $(cpsrc, cpdst)$ where $cpsrc \in AcqS$ and $cpdst \in RelS$. On moving an *acquire* actor above a *release* actor, a new *acquire* – *release* path $(cpsrc, cpdst)$ is added where $cpsrc$ is the *acquire* actor and $cpdst$ is the *release* actor. Therefore, the number of *acquire* – *release* paths is increased by one. ■

Theorem 7.2 An HSDF model G of a task graph TG , transformed to G' by moving an *acquire* actor above a *NSS* actor and a *release* actor, is deadlock-free, provided G is deadlock-free and

- *There is no path from the release actor to the acquire actor involving actors outside the sub-graph in which the acquire and release actors are present or*
- *A path with initial tokens exists from the release actor to the acquire actor involving actors outside the sub-graph in which the acquire and release actors are present.*

PROOF : Given a deadlock-free graph G implies that all the cycles present in this graph have initial tokens. To prove deadlock freedom after moving an *acquire* actor above a *NSS* actor and *release* actor, we should prove that all cycles in G' have initial tokens.

We consider a sub-graph G'_n of G' corresponding to a task T'_n within which *acquire* actors are moved above *NSS* actors. The rest of the graph G' remains intact. The number of possible external cycles is equal to the number of *acquire* – *release* paths in G_n . By Lemma 7.1 and Lemma 7.3 the initial tokens present on external cycles remains intact. According to Lemma 7.2, by moving an *acquire* actor above a *NSS* actor, the number of *acquire* – *release* paths remains the same. However, by Lemma 7.4, by moving an *acquire* actor above a *release* actor, the number of *acquire* – *release* paths has increased by one. This implies the possibility of an additional external cycle in G'_n if there exists a path from the release actor to the acquire actor involving actors outside G'_n . If such a path does not exist, then G'_n is deadlock-free as no additional external cycle is added and all external cycles contain initial tokens. Even if such a path exists, but with initial tokens, then G'_n is deadlock-free as the additional external cycle added contains initial tokens. Similarly, for every sub-graph G'_n of G , the transformation does not cause a deadlock, provided the above conditions are satisfied. ■

An illustration of the proof is provided in Figure 7.10. It shows the HSDF models of a task with two *acquires* and two *releases*, before and after an *acquire* has been moved over a *NSS* and *release*. It can be seen that when *Acq2* is moved over *NSS1* and *Rel1*, a new cycle path (*Acq2*, *Rel1*) is added. This leads to the possibility of an additional external cycle if there exists a path from *Rel1* to *Acq2* through another sub-graph than the one containing these actors. If such a path exists with initial tokens, then G'_n is deadlock-free as the additional external cycle added also contains initial tokens. If such a path does not exist, then G'_n is also deadlock-free as no additional external cycle is added and all external cycles contain initial tokens. The existence of this path can be determined by using path-finding graph algorithms such as depth-first search or breadth-first search.

7.2 Reducing Task-State

By moving *acquires* or combining *acquires*, the task-state can also be reduced. The generation of states inside a task requires that variables that are read and written across states are saved in the task-state. This occurs especially in the case when there are buffers with multiple consumers. The task-state is reduced by moving *acquires* together, so that the variable that is used across multiple states can now be read and written in a single state. However, this may result in a decrease in throughput as the *NSS* corresponding to each *acquire* can now be executed only after all the *acquires* have succeeded. Combining *acquires* has an extra overhead compared to moving *acquires*. This happens because in the case of combining *acquires*, even if one check fails, all the successful checks have to be repeated again. An illustration of the reduction in task-state is shown in the case-study.

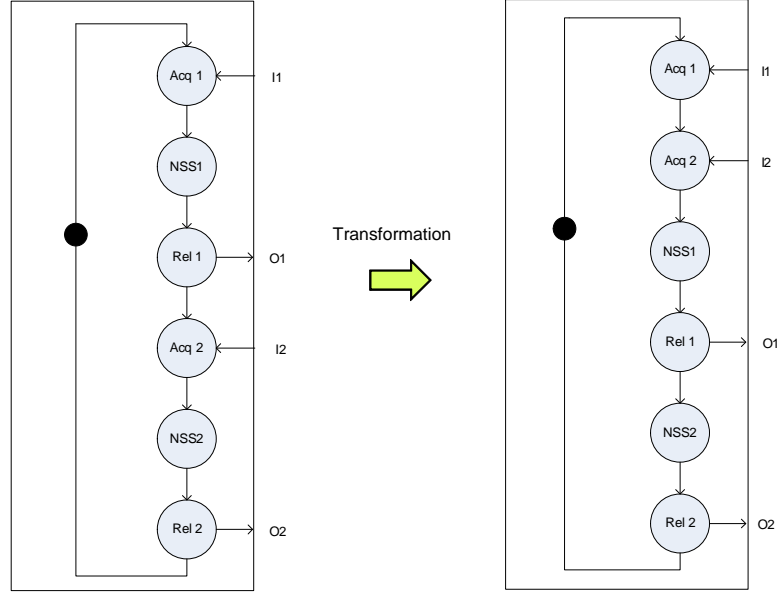


Figure 7.10: Illustration of Proof

7.3 Reducing Synchronization Overhead

Multiprocessor systems using a shared memory require synchronization to ensure correct functional behavior of applications. Especially in applications where the tasks perform little calculation, this synchronization overhead can become dominant [30]. By increasing the synchronization granularity, the synchronization overhead can be reduced, but this often comes at the cost of an increase in buffer capacity. Therefore efforts are being made to reduce the synchronization overhead. An approach is proposed in [31] where adding synchronization edges makes other synchronization edges redundant. If the number of synchronization statements that have become redundant, is higher than the number of added statements, the resynchronization is effective and a reduction of the synchronization overhead is achieved. In order to reduce the synchronization overhead, the number of synchronization statements that acquire data or space in a buffer can be reduced. By reordering *acquires*, a synchronization statement that acquires data or space in a buffer may become redundant. On removing redundant synchronization statements, the synchronization overhead is reduced.

An HSDF model of a task graph is shown in the figure on the left side of Figure 7.11. The HSDF model contains actors corresponding to 3 tasks A, B and C. The task C has been modeled to a finer extent, by a sub-graph G_n showing the *acquire*, *release* and *NSS* actors explicitly. The actors A and B are outside G_n belonging to another task. *Acq1* has a dependency edge from A and *Acq2* has a dependency edge from B. Since actors have positive execution time, it can be seen that a token arrives on edge (B, *Acq2*) later than it arrives on the edge (A, *Acq1*). If *Acq1* and *Acq2* could be reordered, then we obtain an HSDF graph as shown in the figure on the right side in Figure 7.11. Here it can be seen that the synchronization on edge (A, *Acq1*) is redundant because by the time *Acq1* executes, the token would have already arrived, as explained earlier. Hence this redundant synchronization can be removed and it reduces the synchronization overhead.

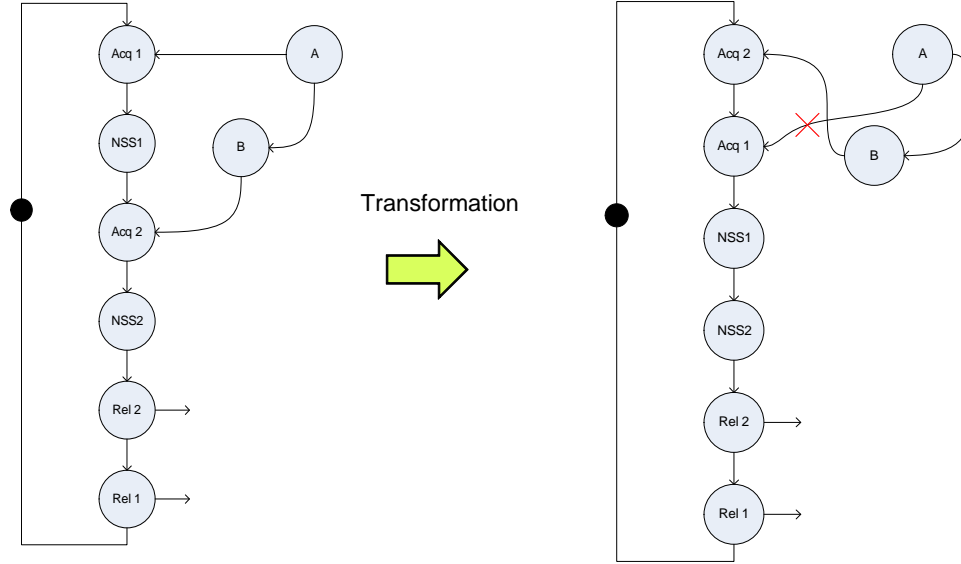


Figure 7.11: Basic Idea - Method I

To identify redundant synchronization edges, the algorithm described in [32] can be applied. However, it can be seen from Figure 7.11 that *NSS1* can no longer execute in parallel with *B* and this may decrease the throughput. An example illustrating this approach is presented in the case-study.

In this approach, one of the actors (*A*) had to have a direct edge with an *acquire* actor (*Acq1*) and a path through at least one another actor (*B*) to another *acquire* actor (*Acq2*). However, such a direct edge does not exist, then if the best case and worse case execution times of actors are available, the following approach can be used to reduce the synchronization overhead. Consider the dataflow model shown in Figure 7.12. Actor *D* has two inputs, each input having a different path from *A*, with at least one actor in the path. Then if the sum of best case execution times of actors in *path1* is greater than or equal to the sum of worst case execution times of actors in *path2*, then data will arrive in the buffer corresponding to edge *B – D*, always later than the buffer corresponding to the edge *C – D*. So the corresponding *acqCheck* for the buffer corresponding to the edge *C – D* can be removed. This reduces the synchronization overhead. However, this approach is only stated here but not detailed further as best-case and worst-case execution times are not available currently for evaluation.

7.3.1 Reordering Acquires

In order to reduce the synchronization overhead one of the *acquires* may have to be moved ahead of the other. This requires that *acquires* should be reordered. Before carrying out this code transformation, it is essential to check that it does not cause a deadlock. Figure 7.13 shows a task before and after reordering the *acquires*.

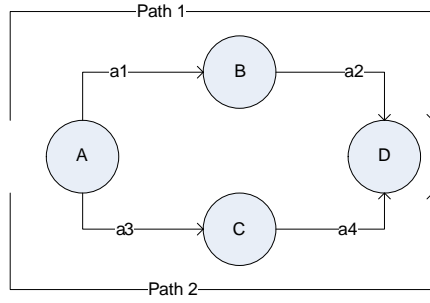


Figure 7.12: Basic Idea - Method II

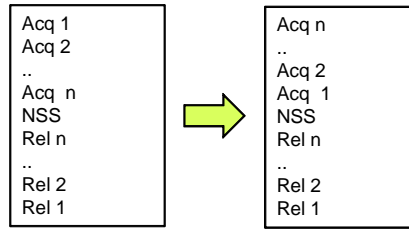


Figure 7.13: Task Transformation

Deadlock Freedom

We prove deadlock freedom by creating an HSDF model of the tasks before and after transformation. Figure 7.14 shows the HSDF model of the tasks shown in Figure 7.13 with two *acquires* and two *releases*. We have actors corresponding to *acquires*, *releases* and NSS in a task.

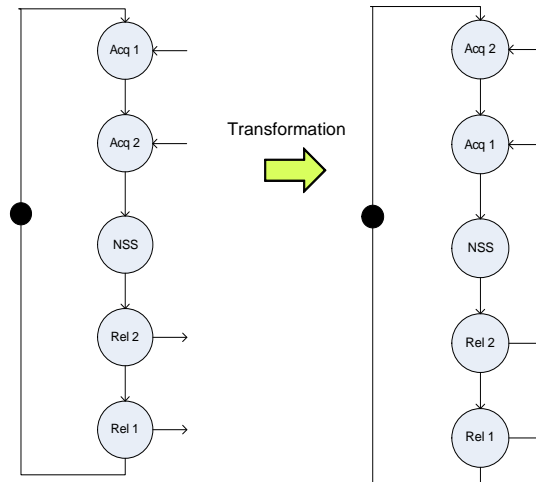


Figure 7.14: HSDF Model

Lemma 7.5 When *acquire* actors are reordered in a sub-graph G_n of G to obtain G_n' , the initial tokens present on corresponding external cycles remain intact.

PROOF : It is given that G is deadlock-free. This implies that all cycles in G have initial tokens. In the sub-graph G_n , there exists an *acquire* – *release* path from every *acquire* actor to every *release* actor (by definition 7.4). No initial tokens are present on an *acquire* – *release* path by construction, as mentioned in definition 7.4. Therefore, all initial tokens on external cycles formed by *acquire* – *release* paths in G_n are present on paths outside G_n . This implies that this transformation does not change the initial tokens present on corresponding external cycles. ■

Lemma 7.6 : When *acquire* actors are reordered in a sub-graph G_n of G , G_n' is obtained wherein the number of *acquire* – *release* paths remains the same.

PROOF : Given a set of *acquire* – *release* paths CP which belong to G_n . We have to prove that after *acquire* actors are reordered, $|CP'| = |CP|$. A *NSS* actor has no dependency edges from any actor outside G_n by definition. By definition 7.4, a cycle path is denoted by $(cpsrc, cpdst)$ where $cpsrc \in AcqS$ and $cpdst \in RelS$. Therefore, though the actual path between $cpsrc$ and $cpdst$ might change upon reordering *acquires*, the number of *acquire* – *release* paths do not change. ■

Theorem 7.3 Assume that an HSDF model G derived from a task graph TG is deadlock-free, then reordering *acquires* in TG to TG' preserves deadlock freedom

PROOF : Given a deadlock-free graph G implies that the all the cycles present in this graph have initial tokens. Assume G' is the HSDF model derived from TG' . To prove deadlock freedom after reordering *acquires*, we should prove that all cycles in G' have initial tokens.

We consider a sub-graph G'_n of G' corresponding to a task T'_n within which *acquire* actors are reordered. The rest of the graph G' remains intact. The number of possible external cycles is equal to the number of *acquire* – *release* paths in G_n . By Lemma 7.6, the number of *acquire* – *release* paths remain the same. This implies that no additional external cycle is added in G'_n because external cycles are formed by *acquire* – *release* paths. By Lemma 7.5 the initial tokens present on external cycles remains intact. Therefore all external cycles of G'_n contain initial tokens and thereby the transformation in G_n does not cause a deadlock.

Similarly, for every sub-graph G'_n the transformation does not cause a deadlock. Therefore all cycles in G' have initial tokens and G' is deadlock-free after this transformation. ■

An illustration of the proof can be seen in Figure 7.15. It shows the HSDF models of a task with two *acquires* and two *releases*, before and after an *acquire* has been reordered with another. The initial tokens are not present on *acquire* – *release* paths $(Acq1, Rel1)$, $(Acq1, Rel2)$, $(Acq2, Rel1)$, $(Acq2, Rel2)$ and hence they remain intact on the external cycles after transformation. It can also be seen that when $Acq2$ is reordered with $Acq1$, some *acquire* – *release* paths have a different path between their source and destination actors. For example, $(Acq2, Rel2)$ has a new actor $Acq1$ in its path. However the number of *acquire* – *release* paths remains the same. Since the number of possible external cycles corresponds to the number of *acquire* – *release* paths and this has not changed, no additional external cycle has been added. Therefore all the external cycles contain initial tokens and hence the transformation does not introduce a deadlock.

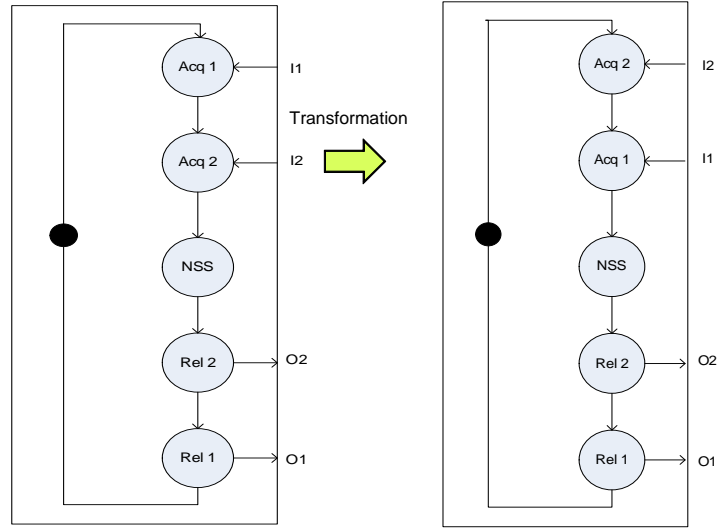


Figure 7.15: Illustration of Proof

Case Study

In this chapter, a case-study is presented to illustrate the applicability of the proposed solutions in this thesis. A DVB-T decoder is used as a case-study. The transformation of buffers with multiple producers and consumers to buffers with only a single producer and consumer is illustrated in this case-study. The generation of FSMs in tasks, is also shown. The possible reduction in states, task-state and synchronization overhead is also illustrated using the case-study.

A DVB-T decoder can be expressed as a sequential NLP in OIL having five modes of operation, namely *IQ*, *ACQ*, *CFO*, *SYNC* and *DECODE*. Initially, the decoder is in the *IQ* mode. In this mode, the decoder estimates the In-phase and Quadrature-phase (IQ) imbalance to obtain the channel response [4]. Once the IQ balance is estimated, the DVBT decoder switches to the *ACQ* mode. Acquisition (ACQ) is performed to find the *OFDM symbol boundary* and the *FFT window size* used by the transmitter. Once these parameters are found, the DVBT decoder switches to the *CFO* mode wherein the Carrier Frequency Offset (CFO) is computed. Once CFO is estimated, the DVBT decoder switches to the *SYNC* mode. In this mode, the DVBT decoder estimates the time tracking parameter e.g. common phase estimation, frequency tracking etc. This estimation is performed by the channel estimation and TPS decoding blocks. Once the parameter estimation is complete, the DVBT decoder switches to the *DECODE* mode in which QAM demapping is carried out. Then the subsequent blocks after QAM demapping, as seen in Figure 1.1 decode the incoming MPEG stream.

For illustration purposes, we use a simplified version of the DVB-T decoder, as shown in Figure 8.1. Here we combine the *IQ* and *ACQ* modes from the more descriptive OIL expression of the DVB-T decoder, into a single mode *ACQ*. Similarly the *CFO*, *SYNC* and *DECODE* modes are combined into a single mode, namely *CFO_SYNC_DECODE*. The *dfe_isr* function returns a data packet, namely *dfe_out*. This data packet is used in the functions present in the above two modes. The decoder remains in the *ACQ* mode until *acq_complete* is asserted. When *acq_complete* is asserted, *frame_boundary*, which corresponds to the OFDM symbol boundary is passed onto the *cfo_sync_decode* function. Subsequently, a transition is made to the *CFO_SYNC_DECODE* mode. The decoder remains in *CFO_SYNC_DECODE* mode, unless the decoding fails and *decode_failed* is

asserted. If the decoding fails, it returns to the *ACQ* mode.

```

task{
    def int x, mode
    def int acq_complete, decode_failed,
    def data dfe_out, frame_boundary;

    mode = ACQ;
    loop{
        dfe_out = dfe_isr();
        x = mode;
        switch(x){
            case ACQ:{
                acq_complete = acq_measure(dfe_out);
                if(acq_complete){
                    mode' = CFO_SYNC_DECODE;
                    frame_boundary = extract_boundary(dfe_out);
                }
                else{
                    mode' = ACQ;
                }
            }
            case CFO_SYNC_DECODE:{
                decode_failed = cfo_sync_decode(dfe_out, frame_boundary);
                if(decode_failed){
                    mode' = ACQ;
                }
                else {
                    mode' = CFO_SYNC_DECODE;
                }
            }
        }
    }while(1);
}

```

Figure 8.1: DVB-T Decoder

The task graph that is generated after parallelization is shown in Figure 8.2. It can be seen that the value of *mode* which is transferred to the variable *x* is required in all the tasks generated from functions inside the *switch* statement. This is because *mode* determines whether each of these tasks should execute or not. The variable *x* in this case has six readers and therefore six buffers have been created, one per producer-consumer pair. Similarly the variables *dfe_out* and *acq_complete* have three corresponding buffers each, one per producer-consumer pair. In the task graph, these buffers with multiple consumers are named with a trailing *unq_cn* where *n* is the number of the consumer. Similarly, buffers with multiple producers are named with a trailing *unq_cn* where *n* is the number of the producer. From Figure 8.1, it can be seen that variable *mode* has five assignment statements writing values to it. Therefore, there are five buffers corresponding to each producer-consumer pair, as can be seen in Figure 8.2.

The *switch* statements in the NLP are transformed to *if – else* statements in the generated tasks. It can be seen that the maximum level of nested *if* statements is two. Therefore, we expect the number of states to be high, because unconditional synchronization has to be performed in every branch. For a task present in the *ACQ* mode in the complete description of the DVB-T decoder, 35 states are obtained. This highlights the difficulty in manually creating an FSM to avoid deadlock.

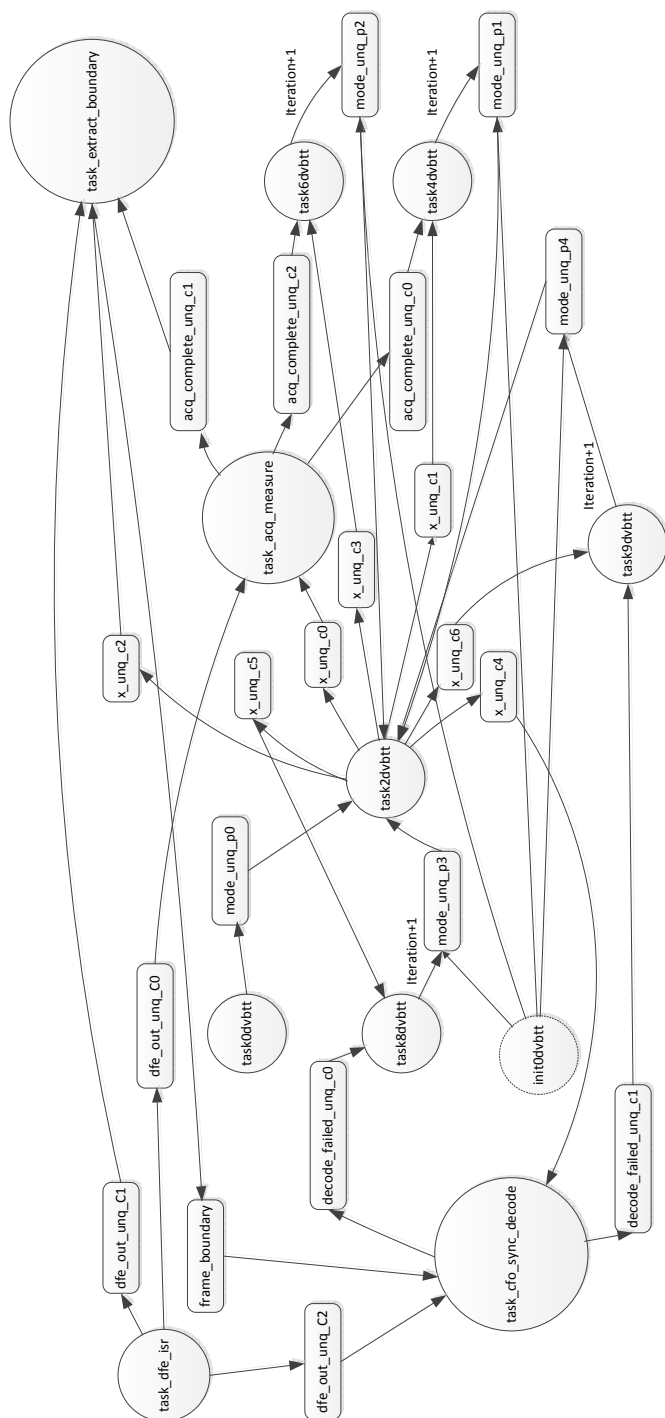


Figure 8.2: Task-Graph

Reducing Number of States

The number of states can be reduced by moving *acquires* out of *if* and corresponding *else* blocks. In order to perform unconditional synchronization, an *acquire* statement is also placed in the block that does not read or write to a buffer which is being accessed in the other block. If these *if – else* blocks do not contain loops, then the *acquire* can be moved just before the *if – else* block and the corresponding *release* can be moved after the *if – else* block. This reduces the number of *acquires*, thereby reducing the number of states.

By combining *acquire* statements in the tasks, the number of states can be reduced. For moving an *acquire* over a *release*, the corresponding HSDF model of the task should be checked, to see if there is path from the *acquire* actor to the *release* actor through any other task. This can be done by path-finding algorithms such as depth-first search and breadth-first search. Currently, this has been performed manually, but it is very tedious and error-prone. The complexity increases with the increase in the number of tasks as more paths have to be traversed. It is considered as future work to automatically create a dataflow model, containing *acquire*, *release* and *NSS* actors, such that the existence of these paths can be checked by applying the above path-finding algorithms.

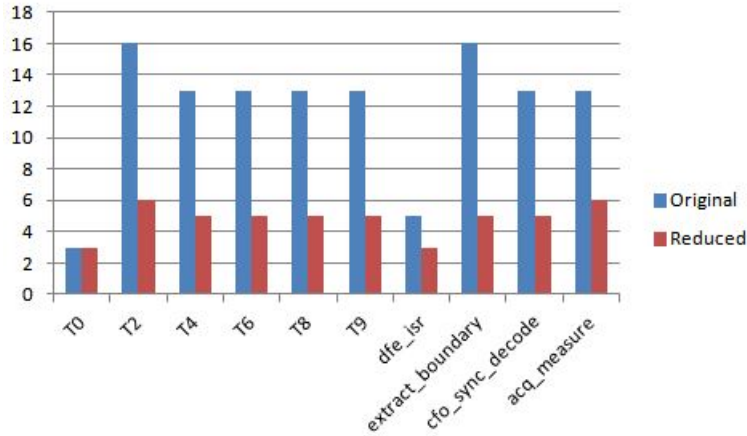


Figure 8.3: Reduction in States

The graph in Figure 8.3 shows the reduction in states in different tasks in the DVB-T decoder. There is a reduction of 60% in the total number of states present in all the tasks. There is a trade-off with the throughput, because *acquires* have been moved above some of the NSS. Therefore, these NSS have to wait until all the *acquires* that were moved above it have succeeded. In order to exploit this trade-off, the reduction in throughput should be analyzed. It is considered as future work to measure the reduction in throughput, and reduce the states accordingly, such that throughput constraints can be met.

Reducing Task-State

By moving *acquires* or combining *acquires*, the task-state can be reduced. This can be seen in the task *dfe_isr*, shown in Figure 8.4. It can be seen that since *dfe_out* is required across multiple states, its value has to be stored in the task-state. By moving

the *acquires* or by *combining* *acquires*, these statements accessing *dfe_out* can be moved together, so that this variable does not have to be stored in the task-state. The task obtained on moving *acquires* can be seen in Figure 8.5. Alternatively, a pointer to *dfe_out* can be stored in task-state, in which case the task-state can still be reduced without combining or moving *acquires*. Combining *acquires* has an extra overhead compared to moving *acquires*. This happens because in the case of combining *acquires*, even if one check fails, even the successful checks have to be repeated again. This is however, not the case with moving *acquires*. When moving or combining *acquires*, there could be a decrease in throughput. This is because *acquires* have been moved above some of the NSS. Therefore, these NSS have to wait until all the *acquires* that were moved above it have succeeded, thereby causing a possible reduction in throughput. The throughput would decrease if the maximum cycle mean (MCM) in the corresponding HSDF model increases by moving or combining *acquires*. The trade-off can between throughput and the reduction in task-state should be exploited, such that throughput constraints can be met. This is considered as future work.

```
ProgressCode_t task_dfe_isr(void){

    if(pState->state == 0){
        acqSpace(dfe_out_unq_c1);
        pState->dfe_out = dfe_isr();
        pState->buffer_dfe_out_unq_c2[pState->c2WriteOffset] = pState->dfe_out;
        relData(dfe_out_unq_c1);
        pState->state = pState->state + 1;
    }
    if(pState->state == 1){
        acqSpace(dfe_out_unq_c2);
        pState->buffer_dfe_out_unq_c1[pState->c1WriteOffset] = pState->dfe_out;
        relData(dfe_out_unq_c2);
        pState->state = pState->state + 1;
    }
    if(pState->state == 2){
        acqSpace(dfe_out_unq_c3);
        pState->buffer_dfe_out_unq_c0[pState->c0WriteOffset] = pState->dfe_out;
        relData(dfe_out_unq_c3);
        pState->state = pState->state + 1;
    }
}
```

Figure 8.4: Task *dfe_isr*

It can be seen that in the case of multiple consumers, there is a reduction of x bytes in the task-state of the producing task, where x is the size of the data being communicated to the multiple consumers. In the case of multiple producers, there is a reduction of $4.(n - 2)$ bytes in the consumer task, where the value four corresponds to the size of the pointer and the n corresponds to the number of multiple producers. In the case of multiple producers, the consumer task identifies which producer wrote to the buffer by checking for *valid/NULL* pointers. These pointers have to be saved across the states. By moving or combining *acquires*, these pointers do not have to be saved in the task-state.

Reducing Synchronization Overhead

The synchronization overhead can be reduced by a reduction in the number of synchronization statements that acquire data or space in a buffer. By reordering *acquires*, some of these synchronization statements may become redundant. A selected

```

ProgressCode_t task_dfe_isr(void){
    if(pState->state == 0){
        acqSpace(dfe_out_unq_c1);
        pState->state = pState->state + 1;
    }
    if(pState->state == 1){
        acqSpace(dfe_out_unq_c2);
        pState->state = pState->state + 1;
    }
    if(pState->state == 2){
        acqSpace(dfe_out_unq_c3);
        dfe_out = dfe_isr();
        pState->buffer_dfe_out_unq_c2[pState->c2WriteOffset] = dfe_out;
        relData(dfe_out_unq_c1);
        pState->buffer_dfe_out_unq_c1[pState->c1WriteOffset] = dfe_out;
        relData(dfe_out_unq_c2);
        pState->buffer_dfe_out_unq_c0[pState->c0WriteOffset] = dfe_out;
        relData(dfe_out_unq_c3);
        pState->state = pState->state + 1;
    }
}

```

Figure 8.5: Task dfe_isr after moving acquires

portion of the the task graph shown in Figure 8.2 is shown in Figure 8.6. The corresponding HSDF model showing only the actors of interest is shown in Figure 8.7. This graph is obtained after reordering the *acquires AcqD frame_b* and *AcqD dfe_out_c2* in the task *cfo_sync_decode*. After reordering, it can be seen that there are two paths from *RelD dfe_out_c2* in the *dfe_isr* task, to *AcqD dfe_out_c2* in the task *cfo_sync_decode*. The shorter path shown in orange between the two actors is redundant as the longer path, which is shown in maroon, enforces the required dependency between the two actors. Such redundant synchronization edges can be found by the algorithm specified in [32].

An *acquireData* function can be split into two functions, *acquireDataCheck* and *UpdateData*, as mentioned in section 7.1.2. The removal of this redundant synchronization edge, means that the corresponding *acquireDataCheck* is not required, and only *UpdateData* corresponding to the update of the counter variables has to be done.

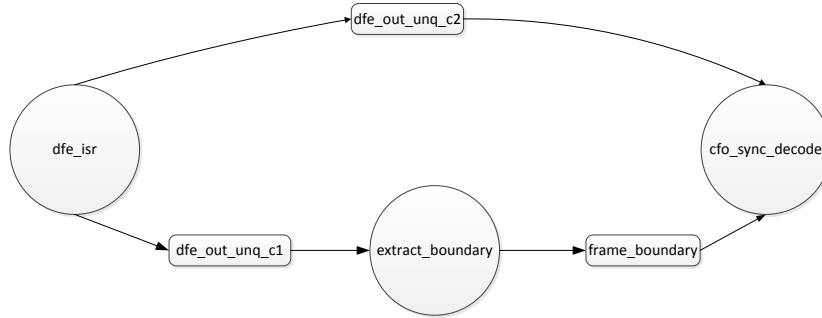


Figure 8.6: Reduction in Synchronization Overhead

An HSDF model corresponding to the task graph given in Figure 8.2 contains 21 *acquireData* synchronization edges. Out of these, six cases have been identified manually, where the *acquireData* synchronization edges may be redundant and can be removed. Thereby it results in a 30% reduction in the number of *acquireData*

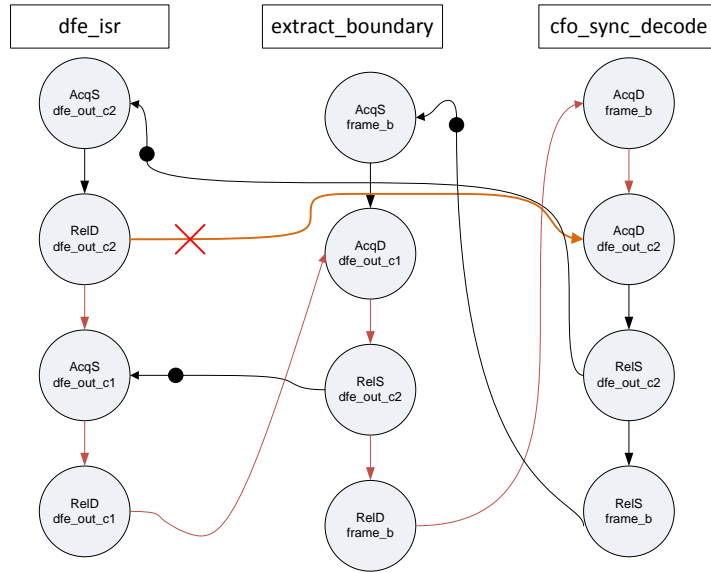
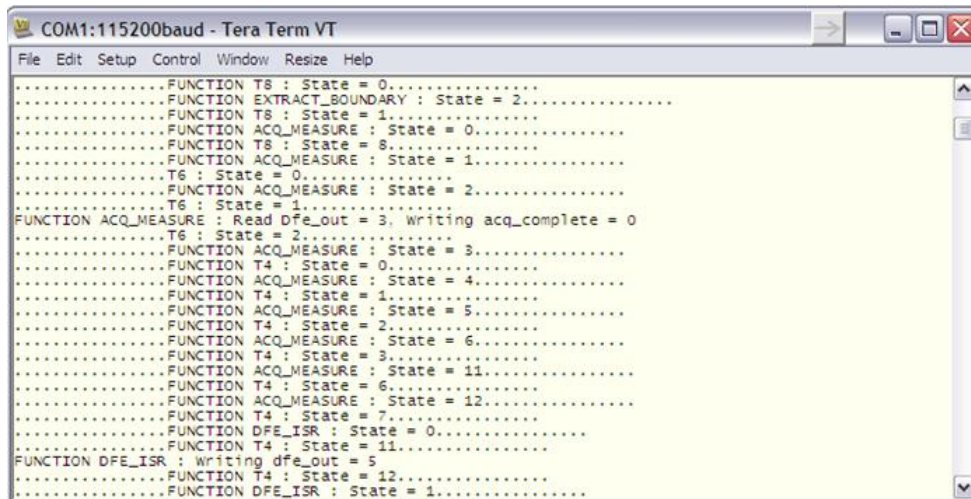


Figure 8.7: Reduction in Synchronization Overhead

synchronization statements. There may be more redundant synchronization statements that are obtained by reordering acquires. However, it is difficult to manually identify them by traversing the HSDF graph.

Execution on MARS

The DVB-T decoder expressed as a sequential NLP in OIL, is used as input to Omphale. A parallelized task graph is created, one for the MARS simulator and the other for the MARS platform. The parallelized task graph is combined with an *Application* file, which creates SoD buffers corresponding to the buffers obtained after parallelization, and connects the ports of these buffers using the SoD API. An executable is created, which is then executed on the NXP MARS multiprocessor platform. A fragment of the output obtained on this execution is shown here for reference in Figure 8.8. The figure shows the transition of the states in the tasks in the DVB-T decoder. This is advantageous for debugging errors as it is possible to see the current state of each of the tasks, and this would ease the debugging effort.



```

COM1:115200baud - Tera Term VT
File Edit Setup Control Window Resize Help
.....FUNCTION T8 : State = 0.....
.....FUNCTION EXTRACT_BOUNDARY : State = 2.....
.....FUNCTION T8 : State = 1.....
.....FUNCTION ACQ_MEASURE : State = 0.....
.....FUNCTION T8 : State = 8.....
.....FUNCTION ACQ_MEASURE : State = 1.....
.....T6 : State = 0.....
.....FUNCTION ACQ_MEASURE : State = 2.....
.....T6 : State = 1.....
FUNCTION ACQ_MEASURE : Read Dfe_out = 3, Writing acq_complete = 0
.....T6 : State = 2.....
.....FUNCTION ACQ_MEASURE : State = 3.....
.....FUNCTION T4 : State = 0.....
.....FUNCTION ACQ_MEASURE : State = 4.....
.....FUNCTION T4 : State = 1.....
.....FUNCTION ACQ_MEASURE : State = 5.....
.....FUNCTION T4 : State = 2.....
.....FUNCTION ACQ_MEASURE : State = 6.....
.....FUNCTION T4 : State = 3.....
.....FUNCTION ACQ_MEASURE : State = 11.....
.....FUNCTION T4 : State = 6.....
.....FUNCTION ACQ_MEASURE : State = 12.....
.....FUNCTION T4 : State = 7.....
.....FUNCTION DFE_ISR : State = 0.....
.....FUNCTION T4 : State = 11.....
FUNCTION DFE_ISR : Writing dfe_out = 5
.....FUNCTION T4 : State = 12.....
.....FUNCTION DFE_ISR : State = 1.....

```

Figure 8.8: DVB-T Decoder on MARS - Console Output

Conclusion

The automatic parallelization tool Omphale, generates parallel tasks from a sequential description of a streaming application, along with the required communication and synchronization statements. In this thesis, techniques are introduced in Omphale, to generate parallel code for a shared-memory multiprocessor platform which uses a non-preemptive scheduler and a communication library with buffers supporting only a single producer and consumer.

In streaming applications, we often encounter *if* and *switch* statements which contain multiple assignment statements in different branches, writing to the same variable, based on the condition. After parallelization, we obtain a buffer with multiple producers, corresponding to this variable. There could also be variables which are read more than once. These would correspond to buffers which have multiple consumers. However, the used communication library, SoD, only supports buffers with a single producer and consumer. In this thesis, we introduced techniques to support buffers with multiple producers and consumers, such that they can use an underlying communication library with buffers, supporting only a single producer and consumer. Moreover, in the case of multiple producers, we need to identify which of the multiple producers has written a value. A technique is introduced using valid/NULL pointers for shared memory systems and a *data valid boolean* for both shared and distributed memory systems. In the case of multiple producers, an alternative is to combine the statements writing to the same variable into a single task. However, this limits the available parallelism.

A sequential NLP can also contain arrays in which the order of accessing its elements is different when reading and writing to it. On parallelization, the array corresponds to a circular buffer with producer tasks writing data to it, in a different order than the order in which the consumer tasks read data. This leads to the reordering problem. Approaches using a FIFO buffer have the overhead of a reordering task and a reordering memory. A sliding windows buffer supports out-of-order access and has windows inside the buffer within which data can be read or written into. We present a modified sliding windows implementation to account for the lack of head pointers of the windows in the buffer administration, by generating corresponding counters, which are stored in the task-state. A wrapper synchronization library is created with blocking *acquire* and non-blocking *release* functions, which use the SoD API functions.

When a non-preemptive scheduler is used with blocking synchronization calls, it can lead to incorrect functional behavior of the application and even deadlock, if the state of the task is not saved. A mechanism is developed to save the state of the task by generating a finite state machine (FSM) inside a task. The extracted tasks generated by Omphale are part of a task graph and each task is organized as a tree containing the statements in a task. The tree contains different types of statements such as *while* loops, *for* loops, *if* statements, assignments, functions etc. The FSM is generated by traversing recursively through the old tree, and creating a new tree of task statements such that the state information is included. The advantage of generating the FSMs in an automatic parallelization tool is that the manual time-consuming and error-prone step of adding states can be avoided.

The generation of states inside a task also requires that variables that are read and written across states are saved in the task-state, such that their values are not lost. This occurs especially in the case when there are buffers with multiple producers or consumers. The size of the task-state is reduced by moving *acquires* together, such that the variable that is used across multiple states can now be read and written in a single state. The number of states is reduced by combining *acquires* together, such that all the corresponding checks for space or data in the buffer are done together. Only if these checks succeed, the corresponding windows are moved forward. However, this may result in a decrease in throughput, thereby creating a trade-off between performance and memory usage. Due to the fine-grained synchronization present in tasks, the synchronization overhead can become dominant. A technique is introduced to reduce the synchronization overhead in acquiring data or space in a buffer, by reordering *acquires* in a task. An attempt at a formal proof showing that the transformations involving moving or reordering *acquires* preserves deadlock-freedom, is presented.

A sequential NLP of a simplified DVB-T decoder is used as a case-study. It is shown that the extracted tasks can be executed on the NXP MARS multiprocessor platform. The number of states and the task-state is reduced by applying the optimizations of moving *acquires* and combining *acquires*. By moving *acquire* statements, the number of states in the tasks of the DVB-T decoder are reduced by 60%. The size of the task-state is reduced by moving or combining *acquires*. The synchronization overhead in acquiring data in buffers is reduced by 30% by reordering the *acquire* statements in tasks.

Future Work

The optimizations proposed required the creation of an HSDF model with *acquire*, *release* and *NSS* actors, from the task graph of the DVB-T decoder. This was done manually and it is tedious and error prone. If the task graph contains a large number of tasks, the corresponding HSDF model would be complex and difficult to create manually. Therefore, it is essential that an HSDF model for a task graph is generated from Omphale. Once an HSDF model has been generated, the proposed optimizations can be implemented in Omphale. An HSDF model cannot be created if a task contains a conditional while loop. An interesting future work is to see how to overcome this limitation such that the optimizations can also be applied to tasks containing such loops.

In order to reduce the size of the task-state, *acquires* can be moved or combined. The number of states in FSMs in tasks can also be reduced by moving *acquires*. When

moving or combining *acquires*, there could be a decrease in throughput because the non-synchronization statements have to wait until all the moved or combined *acquires* have succeeded. This results in a trade-off between reduction in memory usage and throughput. Similarly there is a trade-off between the reduction in states and throughput. These trade-offs can be exploited and the reduction in task-state and number of states in FSMs should be done such that throughput constraints can be met. This is considered as future work.

For moving an *acquire* over a *release* in a task, the corresponding HSDF model of the task graph should be checked, to see if there is a path from the *acquire* actor to the *release* actor through actors not belonging to the same task. Currently, the existence of such paths has been performed manually, but it is very tedious and error-prone. Such paths can be found by path-finding algorithms, if a corresponding HSDF model of the task graph is generated.

Currently, the redundant synchronization edges in the corresponding HSDF model of a task graph, have been found by traversing the HSDF model manually. If an HSDF model is generated, then an algorithm [32] to find synchronization edges can be applied, to find more redundant edges by reordering *acquires*.

Bibliography

- [1] E.A. Lee and T.M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.
- [2] G.J.M. Smit, A.B.J. Kokkeler, P.T. Wolkotte, and M.D. Van De Burgwal. Multi-core architectures and streaming applications. In *Proceedings of the 2008 International Workshop on System Level Interconnect Prediction*, pages 35–42. ACM, 2008.
- [3] T. Bijlsma. *Automatic parallelization of nested loop programs for non-manifest real-time stream processing applications*. Phd Thesis, University of Twente, 2011.
- [4] Y. Jiang, W. Xu, and C. Grassmann. Implementing a dvb-t/h receiver on a software-defined radio platform. *Intl. J. of Digital Multimedia Broadcasting*, 2009.
- [5] A.A. Jerraya, O. Franza, M. Levy, M. Nakaya, P. Paulin, U. Ramacher, D. Talla, and W. Wolf. Roundtable: envisioning the future for multiprocessor soc. *Design & Test of Computers, IEEE*, 24(2):174–183, 2007.
- [6] S. Sriram and S.S. Bhattacharyya. *Embedded multiprocessors: Scheduling and synchronization*, volume 3. CRC, 2009.
- [7] Y. Iosifidis, A. Mallik, S. Mamagkakis, E. De Greef, A. Bartzas, D. Soudris, and F. Catthoor. A framework for automatic parallelization, static and dynamic memory optimization in mpsoc platforms. In *Proceedings of the 47th Design Automation Conference*, pages 549–554. ACM, 2010.
- [8] M.H. Wiggers, M.J.G. Bekooij, and G.J.M. Smit. Buffer capacity computation for throughput constrained streaming applications with data-dependent inter-task communication. In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS’08. IEEE*, pages 183–194. IEEE, 2008.
- [9] L.G. Tesler and HJ Enea. A language design for concurrent processes. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, pages 403–408. ACM, 1968.
- [10] B. Alpern, M.N. Wegman, and F.K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–11. ACM, 1988.

- [11] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, and F. Catthoor. Transformation to dynamic single assignment using a simple data flow analysis. *Programming Languages and Systems*, pages 330–346, 2005.
- [12] 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 *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2011.
- [13] S.J. Geuns. *Parallelization of While-Loops in Nested Loop Programs for Real-time Multiprocessor Systems*. Master Thesis, Eindhoven University of Technology, 2010.
- [14] Software defined radio. http://en.wikipedia.org/wiki/Software-defined_radio. Accessed: 10/09/2012.
- [15] K. Van Berkel, F. Heinle, P.P.E. Meuwissen, K. Moerman, and M. Weiss. Vector processing as an enabler for software-defined radio in handheld devices. *EURASIP Journal on Applied Signal Processing*, 2005:2613–2625, 2005.
- [16] W. Tong. *Channel Decoder Architecture for Mobile Applications*. Master Thesis, Eindhoven University of Technology, 2009.
- [17] A. Turjan, B. Kienhuis, and E. Deprettere. Realizations of the extended linearization model. *Domain-specific processors: systems, architectures, modeling, and simulation*, 2002.
- [18] K. Huang, D. Grunert, and L. Thiele. Windowed fifos for fpga-based multiprocessor systems. In *Application-specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conference on*, pages 36–41. IEEE, 2007.
- [19] T. Bijlsma, M. Bekooij, P. Jansen, and G. Smit. Communication between nested loop programs via circular buffers in an embedded multiprocessor system. In *Proceedings of the 11th International Workshop on Software & Compilers for Embedded Systems*, pages 33–42. ACM, 2008.
- [20] 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 *Systems, Architectures, Modeling, and Simulation, 2009. SAMOS'09. International Symposium on*, pages 140–148. IEEE, 2009.
- [21] J.W. van den Brand and M. Bekooij. Streaming consistency: a model for efficient mp soc design. In *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pages 27–34. IEEE, 2007.
- [22] A. Turjan, B. Kienhuis, and E. Deprettere. Translating affine nested-loop programs to process networks. In *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 220–229. ACM, 2004.
- [23] O.P. Gangwal, A. Nieuwland, and P. Lippens. A scalable and flexible data synchronization scheme for embedded hardware-software shared-memory systems. In *Proceedings of the 14th International Symposium on Systems Synthesis*. ACM, 2001.

- [24] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzter, and G. Essink. Design and programming of embedded multiprocessors: an interface-centric approach. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/Software Codesign and System Synthesis*, pages 206–217. ACM, 2004.
- [25] J. Cockx, K. Denolf, B. Vanhoof, and R. Stahl. Sprint: a tool to generate concurrent transaction-level models from sequential code. *EURASIP Journal on Applied Signal Processing*, 2007(1):213–213, 2007.
- [26] J. Ceng, J. Castrillón, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. Maps: an integrated framework for mp soc application parallelization. In *Proceedings of the 45th annual Design Automation Conference*, pages 754–759. ACM, 2008.
- [27] K. Jeffay, D.F. Stanat, and C.U. Martel. On non-preemptive scheduling of period and sporadic tasks. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 129–139. IEEE, 1991.
- [28] W. Li, K. Kavi, and R. Akl. A non-preemptive scheduling algorithm for soft real-time systems. *Computers & Electrical Engineering*, 33(1):12–29, 2007.
- [29] B.P. Dave and N.K. Jha. Casper: concurrent hardware-software co-synthesis of hard real-time aperiodic and periodic specifications of embedded system architectures. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 118–124. IEEE Computer Society, 1998.
- [30] J.P.H.M. Hausmans, M.J.G. Bekooij, and H. Corporaal. Resynchronization of dataflow graphs. In *Proc. DeAutomation and Ts n Europe Conference and Exhibition (DATE)*, 2011.
- [31] S.S. Bhattacharyya, S. Sriram, and E.A. Lee. Self-timed resynchronization: A post-optimization for static multiprocessor schedules. In *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International*, pages 199–205. IEEE, 1996.
- [32] S.S. Bhattacharyya, S. Sriram, and E.A. Lee. Minimizing synchronization overhead in statically scheduled multiprocessor systems. In *Application Specific Array Processors, 1995. Proceedings., International Conference on*, pages 298–309. IEEE, 1995.