ABSTRACT

TITLE: MODELING AND OPTIMIZATION TECHNIQUES FOR EFFICIENT IMPLEMENTATION OF PARALLEL EMBEDDED SYSTEMS
Ruirui Gu, Doctor of Philosophy, 2010

Dissertation directed by: Professor Shuvra S. Bhattacharyya
and Professor William S. Levine
Department of Electrical and Computer Engineering
University of Maryland at College Park

Embedded systems are becoming more and more important. The products containing embedded systems span from day-to-day household and consumer products, such as digital TVs, mobile phones, and automobiles, to industrial devices and equipment, including, for example, robots, aviation equipment, and high end military and scientific devices such as aircraft. Previously, because embedded systems were highly limited in computational capability, memory size, and power consumption, much research was dedicated to making the best use of limited system resources. In these works, system performance issues, such as execution time, were traded off with system resources, and resources were carefully scheduled and utilized. With more available computational capability in embedded system devices, and more complicated requirements demanding more intensive computation, the most critical design concerns are changing in some important application domains. In such application areas, researchers are paying more and more attention to improving system execution time, which is also the core topic of our work. Execution

time is especially critical to real time systems, in the sense that it is related not only to system performance, but also to system correctness and reliability.

Multi-core devices, which incorporate two or more processors on the same integrated circuits, are becoming increasingly relevant to the design and implementation of embedded systems. In multi-core platforms, carefully managing communication and synchronization among different cores is important to achieve efficient implementations. Two or more processing cores sharing the same system bus and memory bandwidth limit the achievable performance improvements. The ability of multi-core processors to increase application performance depends on the use of multiple concurrent tasks within applications. Therefore, if code is written in a form that facilitates decomposition into concurrent tasks, the multi-core technologies can be exploited more effectively. Dataflow-based languages are suitable for such decomposition into concurrent tasks, particularly in the broad domain of digital signal processing (DSP) applications.

Dataflow representations of DSP software have been explored actively since the 1980s. Such representations have proved to be useful in identifying bottlenecks in DSP algorithms, improving the efficiency of the computations, and designing appropriate hardware for implementing the algorithms.

Dataflow descriptions have been used in a wide range of DSP application areas, such as multimedia processing, and wireless communications. Among various forms of dataflow modeling, synchronous dataflow (SDF) is geared towards static scheduling of computational modules, which improves system performance and predictability. However, many DSP applications do not fully conform to the restrictions of SDF modeling. More general dataflow models, such as CAL [1], have been developed to describe

dynamically-structured DSP applications. Such generalized models can express dynamically changing functionality, but lose the powerful static scheduling capabilities provided by SDF.

This thesis explores modeling and optimization techniques for efficient implementation of parallel embedded systems. We propose a dataflow based framework, which covers modeling, analysis and optimization and bridges between user-friendly design and efficient implementation. The framework is applied to two kinds of applications: control systems and video processing systems.

Model Predictive Control (MPC) has been used in a wide range of application areas including chemical engineering, food processing, automotive engineering, aerospace, and metallurgy. An important limitation on the application of MPC is the difficulty in completing the necessary computations within the sampling interval. Recent trends in computing hardware towards greatly increased parallelism offer a solution to this problem. Our work describes modeling and analysis tools to facilitate implementing MPC algorithms on parallel computers, thereby greatly reducing the time needed to complete the calculations. The use of these tools is illustrated by an application to the critical components of an important class of MPC problems, including the Newton-KKT algorithm, the active set method and linear system solvers.

This thesis also presents an in-depth case study of dataflow-based analysis and exploitation of parallelism in the design and implementation of an MPEG RVC (reconfigurable video coding) decoder. Because dataflow models are effective in exposing concurrency and other important forms of high level application structure, dataflow techniques are promising for implementing complex DSP applications on multi-core systems, and

other kinds of parallel processing platforms. Targeting video processing systems, we use the CAL language as a concrete framework for representing and demonstrating dataflow design techniques. Furthermore, we also analyze our application of the DIF package (TDP), which helps to automatically process regions that are extracted from the original network, and exhibit properties similar to synchronous dataflow (SDF) models. Detection of SDF-like regions is an important step for applying static scheduling techniques within a dynamic dataflow framework. Furthermore, segmenting a system into SDF-like regions also allows us to explore cross-actor concurrency that results from dynamic dependencies among different regions. Using SDF-like region detection as a preprocessing step to software synthesis generally provides an efficient way for mapping tasks to multi-core systems, and improves the system performance of video processing applications on multi-core platforms. Finally the automation from system design to efficient implementation helps our dataflow based modeling and optimization techniques extend into a wide range of embedded applications.

MODELING AND OPTIMIZATION TECHNIQUES
FOR EFFICIENT IMPLEMENTATION OF
PARALLEL EMBEDDED SYSTEMS

By

Ruirui Gu

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2010

Advisory Committee:

Professor Shuvra S. Bhattacharyya, Chair
Professor William S. Levine, Co-Chair
Professor Gang Qu
Professor Andrea Tits
Professor Alan Sussman, Dean's representative

© Copyright by

Ruirui Gu

2010

ACKNOWLEDGMENTS

I owe my gratitude to all the people who have made this thesis possible and because of whom my graduate experience has been one that I will cherish forever.

First and foremost I'd like to thank my advisor, Professor Shuvra S Bhattacharyya for giving me an invaluable opportunity to work on challenging and extremely interesting projects over the past four years. He has always made himself available for help and advice and there has never been an occasion when I've asked a question and he hasn't given me suggestion. It has been a pleasure to work with and learn from such an extraordinary individual.

I would also like to thank my co-advisor, Professor William S Levine. Without his extraordinary theoretical ideas and expertise in the control field, this thesis would have been a distant dream. Thanks are due to Professor Gang Qu, Professor André Tits and Professor Alan Sussman for agreeing to serve on my thesis committee and for sparing their invaluable time reviewing the manuscript.

I want to thank Dr. William Plishker, Dr Mickael Rault and Dr Jorn Janneck for working with me and reviewing some of my papers. Our collaborations and discussions have strengthened this thesis and enriched my research experience.

It has been a pleasure to work and interact with many students in the DSPCAD research group, including Chung-ching, Sankalita, Chia-Jui, Perttu, Sebastian, Hojin, Nimish, Mary, Hsing-Huang, George, and Soujanya.

I feel grateful to all my friends here for their generous friendship, which supported me through the process of completing this thesis.

I would like to give my deepest gratitude to my parents for their love, support, and patience during these years while I focused on studies.

TABLE OF CONTENTS

List of Tables vi

List of Figures viii

1 Introduction 1
- 1.1 Overview 1
- 1.2 Background 5
  - 1.2.1 Dataflow 5
  - 1.2.2 Dataflow Interchange Format 8

2 Methods for Efficient Implementation of Model Predictive Control on Multiprocessor Systems 10
- 2.1 Overview 10
- 2.2 MPC Background 13
- 2.3 Relation of Dataflow to MPC 16
  - 2.3.1 Framework 16
  - 2.3.2 Reactive Control Integrated Dataflow 17
  - 2.3.3 Quadratic Programming and Benchmarks 20
- 2.4 Improved implementations of QP solvers 24
  - 2.4.1 Newton KKT 24
  - 2.4.2 Active Set Method 40
  - 2.4.3 Hierarchical System Model 51
- 2.5 Improved Linear System solvers 52
  - 2.5.1 Gaussian Elimination 53
  - 2.5.2 QR Decomposition 61
- 2.6 Application and Simulation 67

3 Exploiting Statically Schedulable Regions 71
- 3.1 Overview 71
- 3.2 CAL and Scheduling of CAL Systems 75
- 3.3 Analysis framework 78
- 3.4 Derivation of Statically Schedulable Regions 81
- 3.5 Scheduling of SSRs 87
  - 3.5.1 IDCT Example 89
  - 3.5.2 Simulation Results 91
- 3.6 Grouping of Dynamic Ports and SSRs 92

4 Exploring the Concurrency of an MPEG RVC Decoder 97
4.1 Overview . . . 97
4.2 Background . . . 100
4.2.1 Reconfigurable video coding . . . 100
4.2.2 Concurrency . . . 102
4.2.3 Multi-core systems . . . 102
4.3 Inter-actor concurrency analysis . . . 103
4.3.1 Data-driven processing . . . 103
4.3.2 Data parallelism inside CAL networks . . . 105
4.3.3 Pipeline concurrency analysis . . . 106
4.3.4 Concurrency from available code generators . . . 107
4.4 Inter-actor optimization for CAL networks . . . 111
4.4.1 DIF and network analysis capability . . . 111
4.4.2 Interface between DIF and CAL . . . 113
4.4.3 Statically schedulable regions . . . 115
4.4.4 Mapping SSRs into multi-core systems . . . 118
4.4.5 Concurrency analysis of the MPEG-4 SP decoder . . . 124

5 Automatic Integration of SSR and Cal2C 127
5.1 overview . . . 127
5.2 Related work . . . 131
5.2.1 Design Flow . . . 131
5.2.2 XML format . . . 132
5.3 Automated Approach . . . 134
5.3.1 Intermediate Representation . . . 135
5.3.2 Integrating Results of DIF Analysis into the C Back End . . . 136
5.4 The DIFML format . . . 141
5.5 Experimental results . . . 144

6 Summary and Future Work 147
6.1 Framework for Fast Parallel Implementation of Model Predictive Control 147
6.2 Methodology for Quasi-Static Scheduling of CAL Programs . . . 148

Bibliography 151

LIST OF TABLES

2.1 Actor execution times (sec) in Newton-KKT. 27

2.2 Simulation results for group 1. The simulation is conducted 50 times to obtain mean and variance. 37

2.3 Simulation results for group 2. The simulation is conducted 50 times to obtain mean and variance. 37

2.4 Simulation results for group 3. The simulation is conducted 50 times to obtain mean and variance. 38

2.5 Simulation results for group 4. the simulation is conducted 50 times to obtain mean and variance. 39

2.6 Simulation results for group 5. he simulation is conducted 50 times to obtain mean and variance. 40

2.7 The execution time in seconds for different actors in the active set method. 44

2.8 Simulation results of the system before and after applying DPT. 48

2.9 Simulation Results of System with and without MVP. 50

2.10 Simulation results of the system in higher hierarchy. 52

2.11 Simulation results for parallel Gaussian Elimination with different processor patterns. 59

2.12 Simulation results for parallel Gaussian Elimination with different block size. 59

2.13 Simulation results for MPC problems with parallel Gaussian Elimination. 60

2.14 Simulation results for parallel QR decomposition with different processor patterns. 65

2.15 Simulation results for MPC problems with parallel QR decomposition. . . 66
2.16 Average computing time in seconds for aircraft example. 69
2.17 Average computing time in seconds for paper machine model. 70

3.1 MPEG-4 SP decoder performance of CIF sequence 352x288. 96
3.2 MPEG-4 SP decoder performance of sequence 624x352. 96

4.1 MPEG-4 SP decoder performance. 125

LIST OF FIGURES

1.1 Framework and thesis structure: The framework of our methodology is composed of four steps: design (model), analysis, optimization and implementation. The first three steps operate on dataflow graphs. In this thesis, we have applied our framework to two kinds of applications: control systems (using the RCDF model) and video processing systems (using the CALDF model). 3

1.2 A simple example of a dataflow (SDF) model. A, B, C, D, E represent actors. An Edges represents a first-in-first-out (FIFO) queue that directs data values from the output of one actor to the input of another actor. The numbers shown above the edges represent the rates at which actors produce and consume tokens. 8

2.1 Framework for MPC: The dataflow base modeling, analysis and optimization techniques are applied to the application of control systems. This chapter covers the steps and the application indicated as red, bold and italic. 11

2.2 General Structure of Model Predictive Control. *Optimization* is based on the *actual system* and the *model of system*. The result from the optimization problem is then sent back to the *actual system* and the *model of system*. 14

2.3 Dataflow framework for efficient system implementation. 16

2.4 Comparison: traditional framework and dataflow based framework. The dotted line with an arrow represents traditional way from design to implementation. The straight line flow represents dataflow based framework. The step of *design* can be divided into *function specification* and *modeling*. The dataflow based techniques are applied to both *modeling* and *dataflow analysis*. 17

2.5 An example of a regular MTPE and regular MTCE. A, B, C, D, E are actors. e_1, e_2 are regular METC edges that have 1/0 consumption. e_3, e_4 form regular METP edges that have 1/0 production. 20

2.6 RCDF model of the Newton KKT algorithm. There is one set of METP edges: e_2, e_3. The METP parameter $3/0$ indicates that at each iteration, each edge produces either 3 tokens, or none. There is one set of METC edges: e_1, e_2. The METC parameter $3/0$ indicates that at each iteration, each edge consumes either 3 tokens, or none. The unlabeled edge are traditional dataflow edges with normal FIFO property. 25

2.7 RCDF model of Newton KKT algorithm after application of multi-version transformations. 31

2.8 RCDF model of Newton KKT algorithm after transformations. 35

2.9 The original RCDF model of the active set method. There are two sets of MTPEs, and two sets of MTCEs. Here actor M is to check the number of iterations has not exceeded the maximum allowed. 42

2.10 RCDF model of super actor A. Actor A is expanded into a RCDF graph. . 46

2.11 Example of execution before and after the application of DPT. On the left side, before applying DPT, single processor deals with all tokens in a sequential way; On the right side, after applying DPT, parallel processors conduct concurrent computation on a groups of tokens. The number of tokens, indicated as *degree*, is 3. 48

2.12 RCDF model of active set method after application of multi-version transformations. This is parallel version of active set method implementation. 50

2.13 Hierarchical dataflow representation. This is top-level graph, where actor O is a super actor, which can be expanded into a subgraph. 51

2.14 MVP version of hierarchical RCDF model. MVP is applied to actor O. The original actor O in Figure 2.13 is transformed into parallel version. . 52

2.15 Sequential Program of GE. The intensive computations are located in 1.2.1, 1.3.1.1 and 1.3.2. 54

2.16 RCDF model of Gaussian Elimination on a single processing unit. 55

2.17 RCDF model of Gaussian Elimination on four processing units. 56

2.18 2-D Block cyclic distribution of computations onto parallel processing units. After a computation is mapped to the last row or column, the mapping process wraps around cyclically to the first row or column, respectively. 57

2.19 RCDF model of Panal Factorization in one processing unit. P is a super actor which can be expanded into a subgraph including Q, T and J. . . . 63

2.20 RCDF model of Panal Factorization on four processing units. There are four processing units. P is a super actor which can be expanded into subgraph as in Figure 2.19. 64

2.21 Performance comparison of parallel GE and QR. 65

2.22 Complete MPC system of aircraft. 67

3.1 Framework for video processing systems: Analysis. 71

3.2 Outline of method for optimizing dataflow graph implementation. 79

3.3 An illustration of coupled ports and CRGs. 83

3.4 An illustration of weakly connected components. 84

3.5 An illustration of coupled groups. 85

3.6 An illustration of a statically-related group. 86

3.7 An illustration of a statically schedulable region. 88

3.8 SSRs in the IDCT subsystem. 89

3.9 Schedule tree for an SSR in the IDCT example. 91

3.10 IDCT subsystem with a single SSR. 92

3.11 Results: clock cycles vs number of iterations. 93

3.12 A block diagram of an MPEG RVC decoder. 95

4.1 Framework for video processing systems: Optimization. 98

4.2 An RVC block diagram of an MPEG-4 Simple Profile decoder. 105

4.3 CAL2C compilation process: The action translation process starts with an abstract syntax tree (AST) derived from the CAL source code; the transformed CAL AST is expressed in the C intermediate language (CIL) [2], where CAL functional constructs are replaced by imperative ones. 108

4.4 Comparison between direct-C/C++-based implementation and implementation using CAL2C. 110

4.5 Overview of our CAL- and DIF-based method for optimizing dataflow graph implementation. SRP represents statically related port and SSR represents statically related region. 114

4.6 SSR detection in PCG. 117

4.7 SSRs in the IDCT subsystem. 119

4.8 Actor-level mapping onto a multi-core platform. 121

4.9 Scheduling tree for one SSR in the IDCT. 122

4.10 IDCT subsystem with one SSR. 123

4.11 Results: clock cycles vs number of iterations. 123

5.1 Framework for video processing systems: Implementation. 127

5.2 Automation of efficient video processing system generation. 133

5.3 Automated design-to-implementation flow. 134

5.4 Static scheduling: actors row and $transpose$. 138

5.5 Two kinds of statically schedulable regions. 139

5.6 SSR: splitting one CAL actor into two actors. 140

5.7 Code generation procedure. 140

5.8 Experimental results for MPEG4 RVC SP decoder. 145

Chapter 1

Introduction

1.1 Overview

An embedded computer system is a special-purpose computer system designed to perform a small number of dedicated functions, often with real-time computing constraints. The products containing embedded systems span from day-to-day household and consumer products, such as digital TVs, mobile phones, and automobiles, to industrial devices and equipment, including, for example, robots, aviation equipment, paper-making machines, machine tools, and high end military and scientific devices (e.g., aircraft, CAT scanners and ultra-sound machines). Specific applications considered in this thesis include real-time system applications of reconfigurable video coding (RVC) [3] and model predictive control (MPC) [4].

Previously, because embedded systems were highly limited in computation capability, memory size, and power consumption, much research was dedicated to making the best use of limited system resources. Examples include techniques for energy efficient system design [5] and memory size efficiency [6]. In these works, system performance

issues, such as execution time, were traded off with system resources, and resources were carefully scheduled and utilized. With more available computational capability in embedded system devices, and more complicated requirements demanding more intensive computation, the most critical design concerns are changing in some important application domains. In such application areas, researchers are paying more and more attention to maximizing system execution time, which is also the core topic of our work. Execution time is especially critical to real time systems in the sense that it is related not only to system performance, but also to system correctness and reliability.

Benefiting from economies of scale and development of chip technology, there has been a dramatic rise in processing power and functionality since the early applications in the 1960s and what's more, embedded systems have come down in price. Besides the well developed single-processor-on-chip, embedded system designs based on parallel processing units have been emerging in recent years, including multiprocessors and multi-core processors. The latter are attracting more and more attention because of their powerful computation capability and fast synchronization among cores. Our research work explores the systematic exploitation of parallelism in embedded applications, which benefits embedded system performance, reliability, accuracy and correctness.

In general, the process of developing an embedded system is divided into two phases: design and implementation, as shown in Figure 1.1. The system is first designed at a high level of abstraction based on requirements from users and product designers. The high level design is then mapped into an implementation on the targeted processing platform. Our work not only explores dataflow based modeling techniques, but also explores techniques for system-level analysis and optimization that help to bridge the gap

between high-level models and efficient implementations.

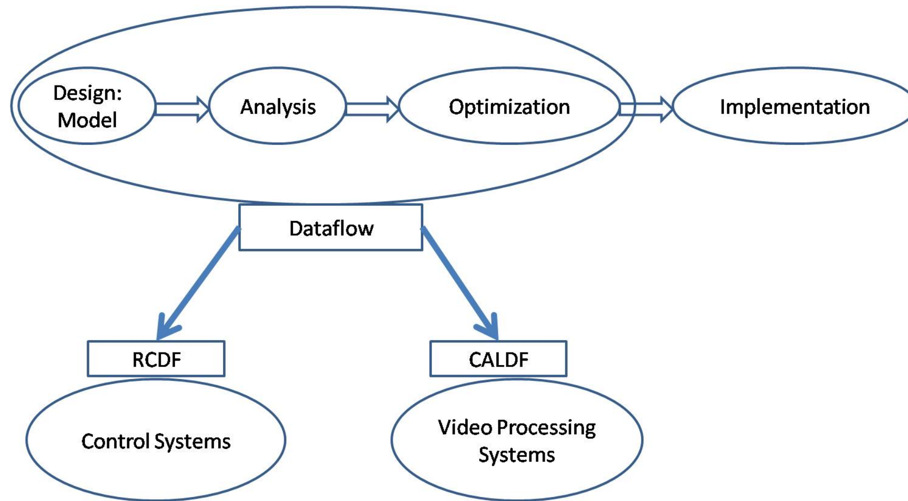


Figure 1.1: Framework and thesis structure: The framework of our methodology is composed of four steps: design (model), analysis, optimization and implementation. The first three steps operate on dataflow graphs. In this thesis, we have applied our framework to two kinds of applications: control systems (using the RCDF model) and video processing systems (using the CALDF model).

Dataflow modeling techniques underlie many popular graphical tools for digital signal processing (DSP) system design (e.g., see [7]). There are different languages and techniques developed in the area of dataflow based design. Our work develops novel methods in the context of the Dataflow Interchange Format (DIF) [8], which is a language for specifying dataflow graphs in terms of subsystems that conform to different kinds of specialized dataflow modeling techniques, and the DIF Package (TDP), which is a tool for analyzing DIF language specifications, with emphasis on scheduling. Although we use DIF and TDP to experiment with and demonstrate our methods, the core methods can be adapted to a wide variety of other dataflow-based design environments — i.e., the underlying concepts are not specific to DIF or TDP.

This thesis focuses on exploring modeling and optimization techniques for efficient

implementation of parallel embedded systems. Figure 1.1 shows how dataflow is related to our work and the overall structure of the thesis. The dataflow based framework, including modeling, analysis and optimization, can be applied to a wide range of applications. Specifically, we have applied it to two kinds of applications: control systems and video processing systems. We have proposed reactive control integrated dataflow (RCDF) to fit the application of control systems, and have applied CALDF [1] for video processing systems.

The rest of the thesis is organized as follows: Chapter 2 describes a general framework called reactive, control-integrated dataflow modeling for analyzing and improving algorithms used for MPC and their hardware implementations. Our work describes modeling and analysis tools to facilitate implementing MPC algorithms on parallel computers, thereby greatly reducing the time needed to complete the calculations and possibly improving their accuracy. The use of these tools is illustrated through application to a class of MPC problems.

Chapter 3 focuses on the detection of SDF-like regions in dynamic dataflow descriptions — in particular, in the generalized specification framework of CAL. This is an important step for applying static scheduling techniques within a dynamic dataflow framework. Our techniques combine the advantages of different dataflow languages and tools, including CAL [1], DIF [8] and CAL2C [9].

Chapter 4 presents an in-depth case study of dataflow-based analysis and exploitation of parallelism in the design and implementation of an MPEG RVC decoder. Because dataflow models are effective in exposing concurrency and other important forms of high level application structure, dataflow techniques are promising for implementing complex

DSP applications on multi-core systems, and other kinds of parallel processing platforms. Furthermore, segmenting a system into SDF-like regions also allows us to explore cross-actor concurrency that results from dynamic dependencies among different regions. Using SDF-like region detection as a preprocessing step to software synthesis generally provides an efficient way for mapping tasks to multi-core systems, and improves the system performance of video processing applications on multi-core platforms.

Chapter 5 proposes an automatic design-to-implementation flow for video processing systems. We present DIFML as the interface between DIF languages and other high level languages. Using XML as a common format to exchange data, we can take advantage of different languages with different features to handle different steps of the framework shown in Figure 1.1. A summary of current progress and future work to explore parallelism is given in Chapter 6.

1.2 Background

1.2.1 Dataflow

Since the mid 1980s, a class of graphical program representations has been evolving steadily, and gaining increasing acceptance among designers of digital signal processing (DSP) systems. Foundations for such dataflow representations have been provided by computation graphs [10], Kahn process networks [11], dataflow architectures [12], and dataflow process networks [13]. Synchronous dataflow (SDF) is a specialized form of dataflow that is streamlined for efficient representation of DSP systems [14].

Since the introduction of SDF, a variety of such *DSP-oriented dataflow models of computation* have been proposed, and DSP-oriented models have been incorporated into many commercial design tools, including Agilent ADS, Cadence SPW (later acquired by CoWare), National Instruments LabVIEW, and Synopsys CoCentric. Useful relationships between dataflow and synchronous languages have also been developed, which helps to connect DSP-oriented dataflow methods to other popular tools, such as Simulink by The MathWorks (e.g., see [15]). Dataflow-based tools for embedded system design use a variety of modeling techniques, and are not necessarily restricted to SDF. These alternative modeling approaches provide different trade-offs among expressive power (the range of DSP applications that can be represented), analysis potential (the rigor with which implementations can be automatically validated or optimized), and intuitive appeal (e.g., see [16]) .

In DSP-oriented dataflow graphs, vertices (*actors*) represent computations of arbitrary complexity, and an edge represents the flow of data as values are passed from the output of one computation to the input of another. Each data value is encapsulated in an object called a *token* as it is passed across an edge. Actors are assumed to execute iteratively, over and over again, as the graph processes data from one or more data streams. These data streams are typically assumed to be of unbounded length (e.g., derived implementations are not dependent on any pre-defined duration for the input signals). In dataflow graphs, interfaces to input data streams are typically represented as *source* actors (actors that have no input edges). An important task when mapping dataflow graphs into implementations is that of sequencing and coordinating among actors based on the resource constraints of the target platform. This task is referred to as *scheduling*.

A simple example is illustrated in Figure 1.2. Here, A and B represent two actors, and the numbers shown above the edges represent the rates at which actors produce and consume tokens. An edge represents a first-in-first-out (FIFO) queue that directs data values from the output of one actor to the input of another. In figure 1.2, actor A produces 2 tokens every time it executes and actor B consumes 3 tokens during each execution. Thus A executes at least two times so that B can execute once. Another example is the edge from actor B to actor C. Every time B executes, it produces two tokens, therefore C can execute two times since it only cosumes 1 token during each execution. How token production and consumption rates are represented, and underlying restrictions imposed on such rates are key distinguishing characteristics of many DSP-oriented dataflow models. In SDF, all data production and consumption rates are restricted to be constant values that are known at design time. The example of Figure 1.2 conforms to the SDF model.

A limitation of SDF and related models, such as cyclo-static dataflow [17] and homogeneous SDF (HSDF) [14], is that dynamic dataflow relationships among computations cannot be described. To express applications that involve such relationships, one must employ models that are more expressive than such *static dataflow* models. Earlier work on DSP-oriented dataflow models has focused heavily on static dataflow techniques, especially SDF. As designers seek to develop more and more complex embedded DSP systems, incorporating more flexible sets of features, and more powerful forms of adaptivity, exploration of dynamic dataflow models is becoming increasingly important.

A variety of dynamic dataflow modeling techniques have been developed previously, including the token flow model [18], stream-based functions [19], enable-invoke dataflow (EIDF) [20], and the CAL actor language [1].

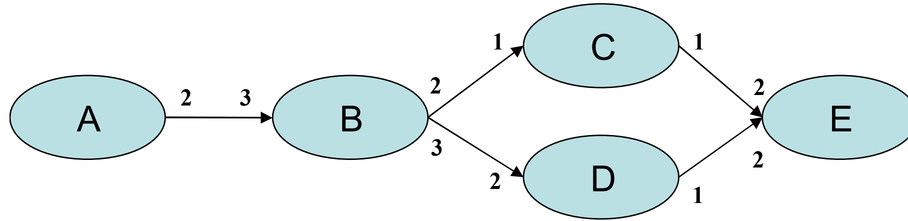


Figure 1.2: A simple example of a dataflow (SDF) model. A, B, C, D, E represent actors. An Edges represents a first-in-first-out (FIFO) queue that directs data values from the output of one actor to the input of another actor. The numbers shown above the edges represent the rates at which actors produce and consume tokens.

1.2.2 Dataflow Interchange Format

The dataflow interchange format (DIF) has been proposed as a standard approach for specifying and integrating arbitrary dataflow-based semantics for DSP system design [21]. The DIF package (TDP) [8, 20] is a software tool, developed in conjunction with DIF, for modeling and analyzing DSP-oriented dataflow graphs. The DIF language (TDL) is an accompanying textual design language for high-level specification of signal-processing-oriented dataflow graphs. The TDL syntax for dataflow graph specification is designed based on dataflow theory and is independent of any specific design tool. For a DSP application, the dataflow semantic specification is unique in TDL regardless of the design tool used to originally enter the specification.

Because dataflow-oriented design tools in the signal processing domain are fundamentally based on actor-oriented design, TDL provides a syntax to specify tool-specific actor information, which ensures that TDP can extract all relevant information from a given design tool [21].

TDL is designed as a standard approach for specifying DSP-oriented dataflow graphs at a high level of abstraction that is suitable for both programming and interchange. TDL

provides a unique set of semantic features for specifying graph topologies, hierarchical design structure, dataflow-related design properties, and actor-specific information. TDP accompanies TDL, and provides a variety of intermediate representations, analysis techniques, and graph transformations that are useful for working with dataflow graphs that have been captured by TDL. Mocgraph is a companion tool that is provided along with TDP. Mocgraph can be viewed as a library of algorithms and representations for working with generic graphs, whereas TDP is a specialized package for working with dataflow graphs. For more details on TDL, TDP, and Mocgraph, we refer the reader to [8, 20].

For example, TDP includes a transformation that converts SDF representations into equivalent homogeneous SDF (HSDF) representations based on the algorithm introduced in [14]. Such a transformation can in general expose additional concurrency [22] that is not represented explicitly in the original SDF graph.

Compared to other design tools for representation and transformation of dataflow graphs — such as SysteMoC [23], PeaCE [24], and stream-based functions [19] — a distinguishing feature of TDP is its support for representing and manipulating different specialized forms of dataflow semantics. This arises from the emphasis in TDL on recognizing a wide variety of important forms of dataflow semantics along with relevant modeling details that are required to meaningfully analyze those semantics. Due to this feature of TDP, its capabilities are highly complementary to those of existing dataflow-based frameworks. In particular, TDL and TDP can be used to capture and analyze, respectively, representations from many of these frameworks.

Chapter 2

Methods for Efficient Implementation of Model Predictive Control on Multiprocessor Systems

2.1 Overview

In this chapter, we apply the framework to model predictive control (MPC), which is one of important applications in the control systems. In this application, our research work covers all the stages in the framework, as shown as red, bold and italic in Figure 2.1. Our work presented in this chapter is also available in the publications [25] [26].

Model Predictive Control (MPC) has found broad application, especially in the process industry. The main limitation on its application is that it is very computationally demanding [27]. As a result, there has been considerable research aimed at speeding up the computation . Most of this research has concentrated on improving the algorithms. Relatively little work [28] has been devoted to improving the implementation of the algo-

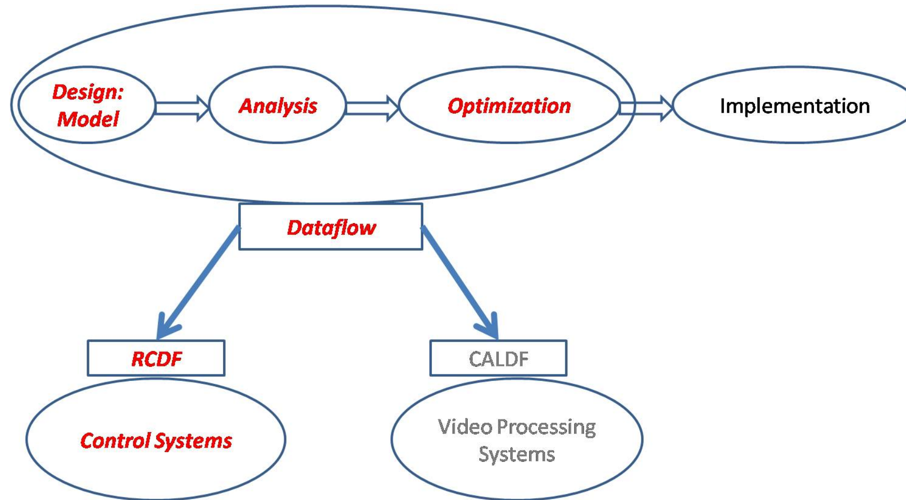


Figure 2.1: Framework for MPC: The dataflow base modeling, analysis and optimization techniques are applied to the application of control systems. This chapter covers the steps and the application indicated as red, bold and italic.

rithms. But the two go hand in hand. For example, Edlund et al. [29] have reduced the time to complete the computations in a specific MPC application by a factor of more than 10 by carefully optimizing the performance of the algorithm.

Recent developments and trends in computing hardware greatly increase the potential for increasing the speed of the MPC computations by properly implementing them in hardware. Specifically, multi-core processes are now prevalent. Dual and quad-core processors are common in today's desktop and laptop computers. Highly parallel and relatively inexpensive processors, such as the Nvidia GeForce 9800 GX2, with 256 stream processors are also available. Because of the inherent trade offs between speed and power consumption in computing the current predictions are that this trend will continue, with the number of cores per processor likely to double every two to three years [30]. Further evidence of this trend is that MATLAB now includes a collection of routines for parallel computation.

It can be very time consuming to analyze code line by line in an effort to find ways to implement it on a parallel machine and to minimize the time required for its execution. Furthermore, it can require considerable expertise to do this effectively. Thus, we are developing an analytical and computational framework to assist the user in doing this optimization. The framework utilizes a high level method for modeling control algorithms. The resulting models display the flow of data and the sequencing of calculations in a way that greatly facilitates their analysis. In particular, it is relatively easy to see where computational and/or storage bottlenecks exist. Once identified, these problems can be eliminated or ameliorated by modifying the algorithm or by proper hardware implementation. Furthermore, the approach is hierarchical. It can be applied to components of the algorithm as well as to the overall algorithm.

Our work on MPC began by exploring efficient implementation of quadratic programming (QP) problems. In the work [25] and [26] we developed faster implementations of the Newton-KKT and active set methods for solving quadratic programming problems. The rationale for doing this first was that most MPC problems are solved by the repeated application of one of these two basic procedures. Thus, fast implementations of these algorithms would benefit almost anyone wanting to apply MPC. Furthermore, we have greatly increased the speed of computation for both Newton-KKT and active set methods by modeling, analyzing, and creating highly parallel implementations of the linear equation solver embedded in both of these algorithms. In order to do this we have had to augment our modeling and analysis tools to include communication delays—an important facet of multiprocessor system performance that should be taken into account carefully when deriving implementations.

In terms of methodology, we developed reactive control dataflow (model) to describe various structures commonly used in the advanced control systems. We further described the basic framework based on the RCDF to model, analyze and optimize the system. In addition, we improved the benchmarks for testing our implementations. This is important because better benchmarks result in more accurate estimates of the time needed for the computations.

2.2 MPC Background

MPC has been studied at least since the 1970s. At that time various works show an incipient interest in MPC in the process industry [31][32]. The basic ideas appearing in MPC are explicit use of a model to predict the process output at future time instants; calculation of a control sequence minimizing a certain objective function; and the application of only the first control signal of the sequence calculated at each step. A detailed introduction to MPC and some specific algorithms can be found in the book [4].

The general structure of MPC is shown in Figure 2.2. The mathematical model is formulated based on the actual system. Optimization problems are derived from the mathematical model, with explicit cost function and constraints. The result from the optimization problem is input to the actual system again to obtain the next state and output. All the MPC algorithms possess common elements and different options can be chosen for each element giving rise to different algorithms.

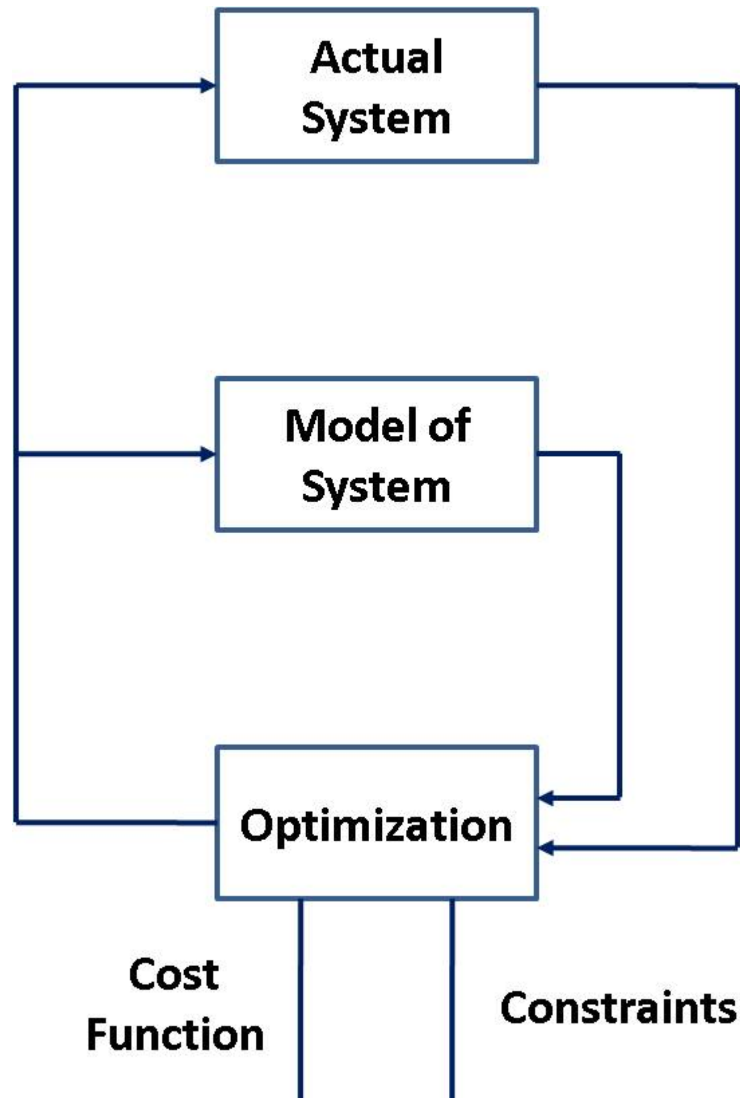


Figure 2.2: General Structure of Model Predictive Control. *Optimization* is based on the *actual system* and the *model of system*. The result from the optimization problem is then sent back to the *actual system* and the *model of system*.

It is well known that MPC can be computation intensive and that, as a result, it can usually be used only in applications with relatively slow dynamics [27]. One approach to addressing this problem has been to compute the control law off-line and store it as a lookup table [33]. However, the situations where this can be done are limited. One would like to be able to compute the controls in real time by solving an optimal control problem. This has prompted a number of researchers to investigate means for increasing the speed with which optimal controls can be computed. Much of this work has focused on improving the algorithms [27, 34].

A few researchers have addressed the implementation of MPC. Ling et al. [35] demonstrated that a “reasonably sized constrained MPC Controller” could be implemented on a modest FPGA chip. Bleris et al. [36] have proposed a computing architecture that is specifically designed for MPC. Furthermore, they have proposed a design framework for application specific processor implementation [28]. Our approach differs from that of Bleris et al. in that we focus on modeling the MPC algorithm structure. This model can be used to derive efficient implementations across a range of architectures. In particular, designers can systematically trade off performance and resource requirements, based on the constraints of the control problem, and the set of available hardware resources.

2.3 Relation of Dataflow to MPC

2.3.1 Framework

The dataflow framework provides a complete solution from system modeling to optimized implementation, as shown in Figure 2.3. First of all, the control algorithm is described as an RCDF model. After all the computation tasks are divided into different actors, we profile the execution time of each actor to determine the bottleneck(s) of the system performance. We then use the dataflow interchange format (DIF) to assist in transforming the dataflow graph into an efficient multiprocessor implementation. DIF provides a design language and associated software tool for experimenting with DSP-oriented dataflow models of computation [8].

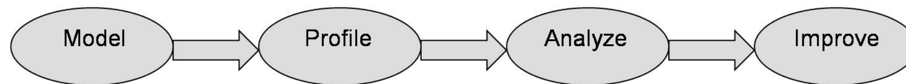


Figure 2.3: Dataflow framework for efficient system implementation.

The further comparison between our methodology with traditional ones is shown in Figure 2.4. In the usual way to develop a control system, implementation follows directly after design, shown as the flow of the dotted line with an arrow going from design to implementation. The control designer usually takes care of functional correctness, and it is hard for him/her to take care of the implementation details as well. Implementation efficiency can be ignored when the controller is relatively simple and computation intensity is not a concern. However, when more complicated algorithms are involved in the controller, implementation efficiency becomes a concern even on high performance pro-

cessors. As shown by a straight line flow in Figure 2.4, we introduce dataflow not only as a modeling tool, but also to assist in the analysis for efficient implementation of the controllers.

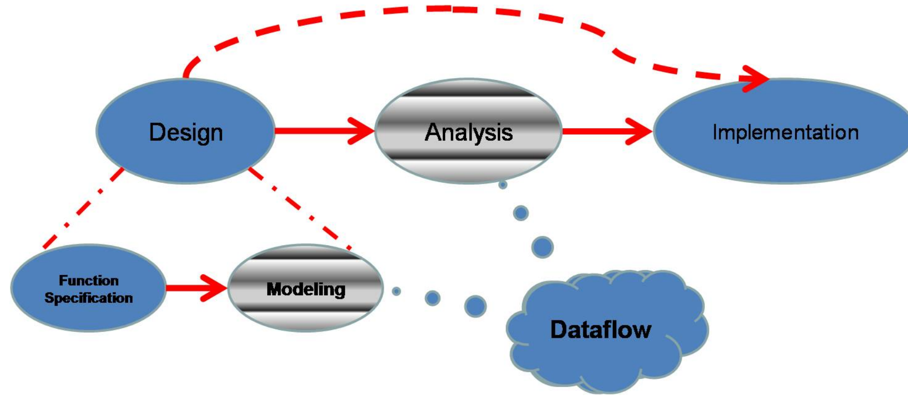


Figure 2.4: Comparison: traditional framework and dataflow based framework. The dotted line with an arrow represents traditional way from design to implementation. The straight line flow represents dataflow based framework. The step of *design* can be divided into *function specification* and *modeling*. The dataflow based techniques are applied to both *modeling* and *dataflow analysis*.

2.3.2 Reactive Control Integrated Dataflow

To facilitate efficient implementation of MPC applications, we introduce a form of dataflow called Reactive Control integrated Dataflow (RCDF), which provides a way to model reactive control structures that are relevant to MPC computations. Reactive Control integrated Dataflow (RCDF) is an extension of SDF, which introduces a way to model reactive control structures. The RCDF model provides a set of mutually-exclusive edges (MEs) and imposes restrictions on the number of tokens produced or consumed on the edge when the source or sink actor, respectively, of the edge executes. Among the MEs, two kinds of special MEs *mutually-exclusive token production edges* (MTPE), and *mutually-exclusive token consumption edges* (MTCE) are especially useful when model-

ing different reactive control structures such as switch and reset.

We first introduce some notation related to dataflow graphs. Given an edge e in a dataflow graph, we denote the source and sink actors of e as $src(e)$ and $snk(e)$, respectively. For a specific edge e_i, and a positive integer j, we use $prd(e_i, j)$ to denote the number of tokens produced by $src(e_i)$ onto e_i during the jth execution of $src(e_i)$, and similarly, we use $cns(e_i, j)$ to denote the number of tokens consumed by $snk(e_i)$ from e_i during the jth execution of $snk(e_i)$. In general, $prd(e_i, j)$ and $cns(e_i, j)$ can be data dependent — i.e., they can depend on the values of samples in the input signals of the dataflow graph. In the restricted case of SDF, such data dependence cannot be present, and furthermore, there can also be no dependence on j — that is, the production and consumption "volumes" must be the same for all values of j.

Given a dataflow graph edge e and a positive integer k, we say that e has *k/0 production* if for all j, $prd(e, j) \in \{0, k\}$. Similarly, a dataflow edge has *k/0 consumption* if for all j, $cns(e, j) \in \{0, k\}$. Notice that by this definition, any edge e in an SDF graph (degenerately) has k'/0 consumption for some $k' = k'(e)$

If S is a set of k/0 production (consumption) edges in a dataflow graph, we employ a minor abuse of notation and refer to S as having k/0 production (consumption).

A major concept in the RCDF modeling approach is that of *mutually-exclusive* production and consumption edges. This concept provides a common framework for representing a useful class of dynamic dataflow structures that is relevant to MPC.

Definition 1: Given a dataflow graph G, a set of *mutually-exclusive token-production edges* (*METP* edges) is a subset $e_1, e_2, \ldots, e_m$ of edges in G such that for any set of input signals applied to G, and for any positive integer j,

$$\sum_{i=1}^{m} I(prd(e_i, j)) = 1, \text{ where } I(x) = \{^{1, x>0}_{0, x\leq 0}$$

represents the indicator function over the positive integers.

Intuitively, a set S of edges is an METP set if across any set of corresponding executions of the edges' source actors, a nonzero token volume is produced on exactly one of the edges in S.

An analogous notion of *mutually-exclusive token-consumption edges* (*METC* edges) can be formulated by replacing the sum in Definition 1 with

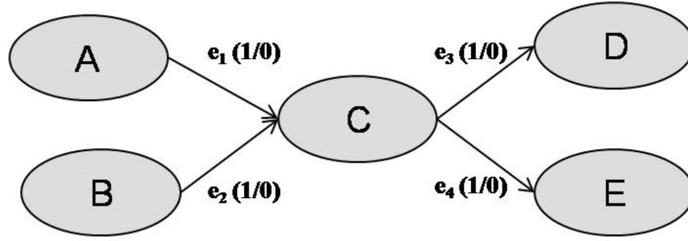
$$\sum_{i=1}^{m} I(cns(e_i, j)) = 1,$$

.

In this chapter, we focus on a particular class of METP edges and METC edges, which we refer to as *regular* METP edges and METC edges. A regular METP is an METP $S = e_1, e_2, \ldots, e_m$ such that all elements have the same source actor ($src(e_i) = src(e_1)$ for all i), and there exists a positive integer k such that S has k/0 production. Similarly, a regular METC is an METC such that all elements have the same sink actor, and there exists a positive integer k such that the METC has k/0 consumption.

Simple examples of regular METP edges and regular METC edges are illustrated in Fig. 2.5. Here, e_1, e_2 are regular METC edges that have 1/0 consumption. At each invocation, either actor A or B (but not both) produces one token on its respective output edge, and these tokens are consumed by C one at a time by successive executions of C. Similarly, each execution of C produces exactly one token across both of its output edges,

so that e_3, e_4 form regular METP edges that have 1/0 production.



METP Edges: {e_1, e_2}; METC Edges: {e_3, e_4}.

Figure 2.5: An example of a regular MTPE and regular MTCE. A, B, C, D, E are actors. e_1, e_2 are regular METC edges that have 1/0 consumption. e_3, e_4 form regular METP edges that have 1/0 production.

2.3.3 Quadratic Programming and Benchmarks

We demonstrate our work to improve the system performance by simulation of two kinds of benchmark problems.

The first kind of problems is abstracted into randomly generated matrices. The object function is formulated using these randomly generated matrices. Using this kind of problem, we demonstrate that RCDF works on general optimization problems. Consider the following QP problem $< P_0 >$, in standard inequality form:

$$\text{minimize} \, \frac{1}{2} < x, Qx > + < c, x >, \text{ s.t. } Ax \leq b, x \in R^n$$

with $A \in R^{m \times n}$, $b \in R^m$, $c \in R^n$, and with $Q \in R^{n \times n}$ symmetric. In order to solve this problem using the Newton KKT method, we introduce some notation. Let $I = 1, ...m$, where m is the number of rows of A, and, for $i \in I$, let a_i be the ith row of A, let b_i be

the ith entry of b, and let $g_i(x) \equiv < a_i, x > -b_i$. Also let $f(x) \equiv \frac{1}{2} < x, Qx > + < c, x >$. The matrix Q, with condition number 10^{ncond} and number of negative eigenvalues approximately $negeig$, was generated as described in [37]. The vector c is defined as $c = -Qx^*$, where x^* is chosen from the normal distribution $N(0, 1)$. The upper part of A, $n \times n$, is a diagonal matrix with diagonal entries as -1s. The entries of the lower part of A are chosen independently from a uniform distribution on the interval $(10^{-6}, 1 + 10^{-6})$. The algorithms are initialized with $x^0 = e$, where e represents the vectors of all ones. b is selected as a vector with the first n elements as 0 and the rest as $b = Cx^0 + e$. In the simulation, we choose $n = 100$.

The second kind of problem is related to practical MPC applications. These are benchmarks we generated for MPC applications. Generation of benchmarks is an interesting direction to study further in the future.

In practice, many MPC problems involve repeated solutions of:

$$\text{minimize} \sum_{k=0}^{N-1} (x^{'}(k) C^T C x(k) + u^{'}(k) u(k)) + x^{'}(N) C^T C x(N)$$

s.t.

$$x(k+1) = Ax(k) + Bu(k), k = 0, \cdots, N-1$$

$$|u_i(k)| \leq u_{max}, i = 1, \cdots, m, k = 0, \cdots, N-1$$

$$x(0) = x_0, x_0 \text{ is constant}$$

Here A is an $n \times n$ matrix, B is an $n \times m$ matrix, and C is a $p \times n$ matrix. $x(k)$ is an $n \times 1$ vector, and $u(k)$ is an $m \times 1$ vector. For simplicity of notation it has been assumed

that all the controls are weighted equally. This assumption can be trivially relaxed.

In order to create a family of benchmark problems to use in evaluating and testing our implementations of MPC, we randomly chose 50 values for the three matrices A, B, and C, all sets with $n = 10$, $m = 8$, and $p = 8$. We then checked whether (A, B) was controllable. If not, we deleted that trio A, B and C from the set. If they were controllable we then checked if (A, C) was observable. If not, then we deleted that A, B and C. The remaining trios of matrices constitute a collection of test problems of randomly varying computational difficulty. To complete the problem formulation, we chose $N = 50$.

In order to reduce the resulting MPC problems to a form in which the Newton-KKT or active set methods can be easily applied, we formed the large matrices given below:

$$\hat{A} = \begin{bmatrix} B & 0 & \cdots & 0 \\ AB & B & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots \\ A^{N-1}B & A^{N-2}B & \cdots & B \end{bmatrix},$$

$$\hat{C} = \begin{bmatrix} C'C & 0 & \cdots & 0 \\ 0 & C'C & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & C'C \end{bmatrix},$$

$$\hat{d} = \begin{bmatrix} A \\ A^2 \\ \cdots \\ A^N \end{bmatrix} * x_0,$$

The result is the quadratic programming (QP) problem $< P_1 >$, which is similar to $< P_0 >$:

$$\text{minimize } (\hat{A}\hat{u} + \hat{d})\hat{C}(\hat{A}\hat{u} + \hat{d}) + \hat{u}^T\hat{u}$$

subject to

$$|u_i(k)| \leq u_{max}, i = 1, \cdots, m, k = 0, \cdots, N-1$$

where

$$\hat{u} = \begin{bmatrix} u(0) \\ u(1) \\ \cdots \\ u(N-1) \end{bmatrix}$$

Note that each of the $u(k)$ is an m-vector so the overall dimensions of $\hat{u}$ are $Nm \times 1$.

Next we will provide a detailed description of how the dataflow-based framework applies to different solution algorithms for MPC.

2.4 Improved implementations of QP solvers

2.4.1 Newton KKT

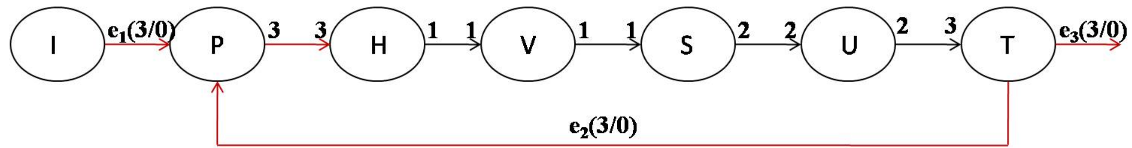
In numerical analysis, Newton's method is one of the best known methods to find successively better approximations to the roots of a real-valued function. Newton's method, as an optimization algorithm, is also well-known for finding the value of $x \in R^n$ that minimizes a twice-differentiable function $f : R^n \to R$.

The Karush-Kuhn-Tucker conditions (KKT) are necessary for a solution of a nonlinear programming problem to be optimal provided some regularity conditions are satisfied.

The Newton-KKT method takes advantage of both Newton's method and the KKT conditions. Newton-KKT methods are algorithms in which search directions for the primal variables and the KKT multiplier estimates are components of the Newton (or quasi-Newton) direction for the solution of the equalities in the first-order KKT conditions of optimality or a perturbed version of these conditions. The specific version of Newton KKT we use was introduced by Absil and Tits [38]. Their methods are adapted from previously proposed algorithms for convex quadratic programming and general nonlinear programming. The Newton-KKT algorithm is a good choice for use in solving the discrete-time optimal control problems that are central to MPC.

The Newton KKT algorithm to solve the problem $< P_0 >$ is modeled by the RCDF model in Figure 2.6. We implement communication between actors based on the dataflow model. However implementation of each actor follows the sequential program-

ming method. As shown in Figure 2.6, there are seven actors in the system, and each actor is responsible for a specific function. The function of each actor is described in brief as follows:



METP Edges: {e_2, e_3}; METC Edges: {e_1, e_2}.

I—InitializeValue; P—ParameterCalculation; H—HessianCalculation;
V—MiddleValueCalculation; S—SearchDirection; U—Update; T—StoppingCriterion;

Figure 2.6: RCDF model of the Newton KKT algorithm. There is one set of METP edges: e_2, e_3. The METP parameter $3/0$ indicates that at each iteration, each edge produces either 3 tokens, or none. There is one set of METC edges: e_1, e_2. The METC parameter $3/0$ indicates that at each iteration, each edge consumes either 3 tokens, or none. The unlabeled edge are traditional dataflow edges with normal FIFO property.

I—The actor I is used to initialize the values of state variables and the values of parameters, such as the tolerance threshold, that are used later.

P—The actor P is used to compute the values of f, g and the Schur complement at the current value of x for every iteration.

H—The actor H is used to compute the modified Hessian matrix. It functions only under the condition that the Hessian matrix has one or more negative eigenvalues.

V—The actor V is used to compute the gradient of f in every iteration.

S—The actor S is used to compute the search direction for the next iteration. It finds the solution by solving a linear system of equations.

U—The actor U is used to compute the updated values of x, f and g.

T—The actor T is used to compare the difference between the updated value and

previous value with a given criterion, to see if the system needs to go to the next iteration or terminate in this iteration.

There is one set of METP edges: e_2, e_3. The METP property indicates that at each iteration, there is only one edge producing tokens. In other words, either e_2 or e_3 produces tokens at one iteration. The METP parameter $3/0$ indicates that at each iteration, each edge produce either 3 tokens, or none. There is one set of METC edges: e_1, e_2. The METC property indicates that at each iteration, there is only one edge consuming tokens. In other words, either e_1 or e_2 consumes tokens at one iteration. The METC parameter $3/0$ indicates that at one iteration, each edge consumes either 3 tokens, or none.

Since the algorithm is decomposed into actors based on functionality, the code size and complexity generally varies across the actors. This is typical of dataflow-based program representations. Some of the more complex actors, most notably U, may represent *hierarchical actors* whose internal functionalities are described by additional ("nested") dataflow graphs. We elaborate on the internal representation of actor U in the following section.

2.4.1.1 Profiling and Identification of Bottlenecks

Based on our dataflow-based modeling approach, along with MATLAB implementations of the individual actors, we have conducted MATLAB simulations to evaluate the contribution of each actor to the overall execution time required for the application. In our analysis and use of execution time information, we have ignored certain "fine-grained" actors that have very low computational cost. For example, we have ignored actor I, which is used only to initialize parameters. We have also ignored the execution time

Table 2.1: Actor execution times (sec) in Newton-KKT.

$ncond$	0		3		6	
statistics	mean	variance	mean	variance	mean	variance
H	0.01223	4e-5	0.01076	5e-5	0.01246	4e-5
S	0.00815	6e-5	0.00564	5e-5	0.02586	7e-5
U	0.00475	5e-5	0.00487	5e-5	0.01023	5e-5
V	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000

contributions of actors P, V and T, which involve simple addition and equality-testing operations.

In our MATLAB simulations, the matrix Q was generated based on the condition number ($ncond$) parameter, and the number of negative eigenvalues ($ngeig$). We fixed the parameter $ngeig$ to be 5, and chose $ncond$ from the set $\{0, 3, 6\}$. For each $ncond$ value, we execute 50 times. The statistical results, $mean$ and $variance$, are calculated based on the simulation. Table 2.1 shows the execution times of different actors for different values of $ncond$ and otherwise randomly chosen Q. These values were determined by implementing each actor in MATLAB (version 7.04), executing the code for 10 random choices of Q, and recording the mean and variance of the time required to complete the computations.

It is easily seen from Table 2.1 that actors H, S, and U impose a relatively high computational load. The computation time spent on V is so little that it is ignored here.

In the next section, we describe how our dataflow-based model together with its profiled execution time information can be used to strategically dedicate parallel hardware resources and accelerate the computation of performance-critical components in the overall design.

2.4.1.2 MULTI-VERSION PARALLELISM

Two important forms of parallelism that are exposed effectively by dataflow graphs are functional parallelism and data parallelism. Functional parallelism refers to the simultaneous execution of distinct actors on separate hardware resources, whereas data parallelism involves simultaneously executing the same actor on separate resources.

A hybrid form of parallelism, which we call *multi-version parallelism*, can be very useful when hardware resources are relatively abundant and constraints on performance are relatively stringent. We view multi-version parallelism as a hybrid form because it relates to aspects of both functional and data parallelism. As an example of multi-version parallelism, consider the search direction calculation actor in the NEWTON-KKT example of Figure 2.6. Two different algorithms are given in [38] for determining the search direction. The first, which we denote by actor S_1, is an augmented system. In some applications, a normal system, also given in [38] converges faster. We denote it by actor S_2.

The time required to complete an execution of S_1 or S_2 is in general data-dependent, and the relative speeds of corresponding executions (i.e., executions that have the same "j" index) are also data-dependent. In general, for some values $j \in J_1$, each jth execution of S_1 will complete before the jth execution of S_2, and for other values $j \in J_2$ ($J_1 \cap J_2 = \emptyset$), the jth executions of S_2 will complete sooner.

A multi-version implementation of the search direction calculation based on alternative implementations S_1 and S_2 therefore involves executing them both in parallel (simultaneously on separate resources), and taking the result of the S_i that finishes first. As

soon as one of the "versions" completes, its result is taken as the result of the corresponding execution of the search direction calculation, and the current execution of the other version is terminated. Such a multi-version implementation is useful whenever there can be significant variation between which of the versions completes first, and the available hardware resources accommodate parallel execution of the different versions.

Parallelism can also be taken advantage of to improve accuracy while ensuring reliability in a flexible way. For a specific task, one could also allocate a fixed maximum time to the calculation of the task. If, for example, S_1 was usually faster but less accurate than S_2, we could then wait for S_2 to complete even though S_1 has already finished. In this way, we could achieve better accuracy within a reasonable time. If the maximum time has expired before S_2 has completed and after S_1 has completed, you would take the result from S_1. In this way, reliability is guaranteed for the system. If neither S_1 nor S_2 terminates within the admissible time frame, it might still be possible to choose the better of the two options. We elaborate the mechanism of *handling failed hard read time requirements* in [25].

If one replaces an actor X with actors $x_1, x_2, \ldots, x_n$ that represent multiple versions of X, then with respect to the new (transformed) dataflow graph, parallelism among $x_1, x_2, \ldots, x_n$ can be viewed as functional parallelism. Multi-version implementation is also related to data parallelism since the parallel executions of $x_1, x_2, \ldots, x_n$ operate on one or more common data streams in the original dataflow graph (the data streams associated with the input edges of X). Thus, in some sense, we can view multi-version parallelism as a hybrid form of parallelism that involves aspects of both functional and data parallelism.

2.4.1.3 Evaluating Newton-KKT implementation

Through our execution time analysis on our model of Newton-KKT, we found, as described in Section 2.4.1.1, that the major computational bottlenecks are the actors H, S, and U.

Multi-version implementation of H

To alleviate the bottleneck due to H (Hessian calculation), we apply a multi-version implementation of H. Two different methods for adjusting the Hessian to be positive definite are mentioned in [38]. The first, H_1, is more reliable while the second, H_2, is usually faster. The new dataflow graph that results from applying the multi-version transformation to H and S is illustrated in Fig. 2.7. Here, H_1 and H_2 represent the computation-every-iteration and vector-based-computation methods, respectively. The detailed description of each actor is as follows:

- I: InitializeValue
- P: ParameterCalculation
- H_1: HessianCalculationOption1
- H_2: HessianCalculationOption2
- R: MVOS
- V: MiddleValueCalculation
- S_1: SearchDirectionOption1

- S_2: SearchDirectionOption2
- U: Update
- T: StoppingCriterion

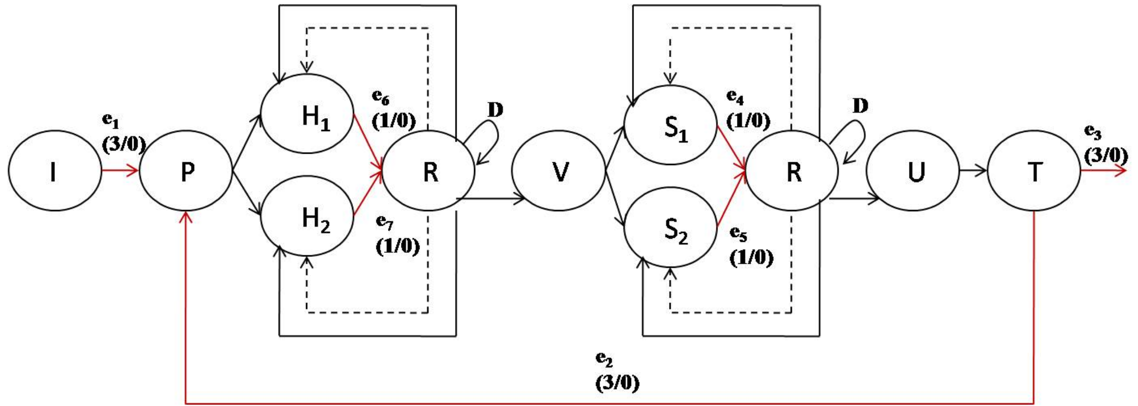


Figure 2.7: RCDF model of Newton KKT algorithm after application of multi-version transformations.

Actor R in Figure 2.7 represents a special actor, which we call a *multi-version output selector* (*MVOS*), for multi-version implementation. R is an RCDF actor that samples its input edges in some pre-defined order. As soon as it finds an input edge that contains a token, it reads the token and copies it to its output. The actor then samples the remaining inputs and discards any tokens that it finds on those inputs — this happens in the event that different versions of the associated multi-version actor have produced their outputs at relatively closely-spaced points in time. After any such "redundant" inputs have been discarded, an MVOS actor sends a single token to each of the separate "version" actors (H_1 and H_2 in this example) to enable the next invocations of these actors. The

use of such enabling tokens ensures that at most one execution of each version actor is allowed at any given time (i.e., version-level, data-parallel operation is prevented). Use of these enabling tokens can be "optimized away" if other details in the implementation preclude data parallel operation — for example, if each version actor is mapped to a single hardware resource that is capable of performing only one version execution at any given time.

The MVOS can also optionally send an asynchronous reset signal to each version actor. These signals are asynchronous in the sense that they are not synchronized with dataflow firings (executions) of the actors that they are controlling. Such reset signals are useful, for example, to save execution time and power consumption associated with version executions that are ignored because another version has "won the race" already for the current execution. Such a signal can be implemented through a software interrupt or through an asynchronous hardware reset logic, depending on the type of implementation platform. The asynchronous reset signals generated by the MVOS actor R are represented by dashed lines in Figure 2.7. The main drawback associated with using these reset signals is that they deviate from pure dataflow semantics, and this may complicate certain forms of analysis of the overall specification. Some of the traditional dataflow analysis techniques can not be applied to these reset signals directly. Studying ways to systematically integrate such asynchronous reset signals into DSP-oriented dataflow graph analysis is a useful direction for further study.

The *self loop* edge of R (the edge whose source and sink are both R) represents the state variable used by R that determines whether 1) the actor is presently monitoring its input edges to determine which version has won the race for the current execution, or 2)

the actor is in a state of discarding input tokens because the race is over. The D next to this edge represents a unit *delay*. Such delays represent initial tokens on edges.

Handling failed searches

Multi-version implementation is especially attractive for scenarios in which complex searches must be carried out. In practice, such search techniques can sometimes fail to find solutions within the allowable period of time that can elapse before a response must be produced by the system. In such cases, if the system keeps waiting, the whole system may stop or "crash", leading to disastrous or otherwise undesirable consequences. This is important in MPC because the optimization computation may fail to converge in the available time.

To prevent such failures, timers can be incorporated into MVOS actors so that whenever a timeout occurs, asynchronous reset signals, and next-execution-enable tokens are sent to all of the associated versions. In such cases, the MVOS actor can respond with a copy of the value that it produced by the previous execution of the corresponding set of version actors. More elaborate approaches for handling timeout problems in multi-version actors are worthy of further investigation.

Transformation of U

To alleviate the bottleneck due to U, we examined the MATLAB source code for U and replaced it with the equivalent, hierarchical ("nested") RCDF graph shown in Figure 2.8. The detailed description of each actor is as follows:

- U_{t1}: CalculateAllEntryValue

- U_{t2}: FindMinimalOfSet
- U_{t3}: CalculateValueOfBar
- U_{t4}: InitializeValueOfBar
- U_{t5}: UpdateNextValue

We performed this RCDF-to-MATLAB transformation manually, through our understanding of the algorithm and relationships among its various parts. Because of the high expressive power of MATLAB, automated conversion of MATLAB to specialized DSP-oriented dataflow representations is in general undecidable (computationally infeasible), although investigation of such conversion under restricted cases is an interesting direction for further study.

Our refinement of actor U as a nested RCDF graph exposes opportunities for exploiting functional parallelism among actors U_{t2}, U_{t3}, and U_{t4}. Also, the U_{t1} and U_{t2} actors are both well-suited to exploiting data parallelism because they involve relatively simple operations that are applied independently to successive items of data in their respective input streams. In our experiments, we have exploited both the functional parallelism and data parallelism described above that is associated with our RCDF refinement of actor U. These individual actors and experiments are described in Section 2.4.1.3.

Experimental Results

To demonstrate our parallel processing methods for Newton KKT, we have evaluated them with MATLAB simulations. We use the first kind of benchmarks. Consider the QP problem $< P_0 >$, we generate the matrix Q by randomly choosing two parameter.

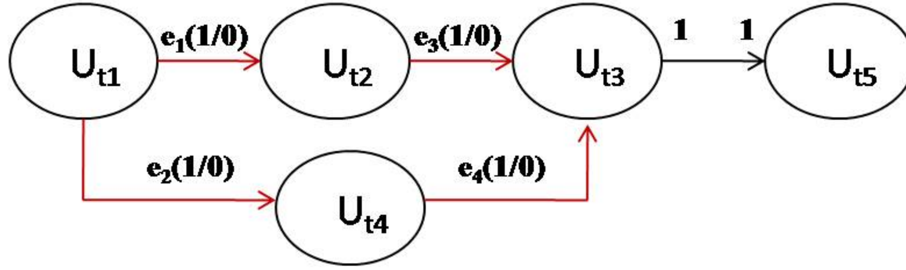


Figure 2.8: RCDF model of Newton KKT algorithm after transformations.

One is condition number, indicated as *ncond*, and the other is number of negative eigenvalues approximately, indicated as *negeig*. The algorithm were tested on a dell desktop with Intel Pentium 4 CPU at 3.0GHz. Each benchmark was executed 50 times and then we calculated the values of $mean$ and $variance$. These simulations take into account the detailed functionality of each actor, and our analysis of the simulation results provides estimates for the performance improvements gained through our integrated application of RCDF modeling, multi-version transformations, and hierarchical refinement.

Our simulations take three forms:

1. Sequential version: all actors execute sequentially, as they would execute in a conventional, single-processor implementation.

2. Parallel, multi-version (PMV): multi-version parallelism is exploited for actors H and S, as described in Section 2.4.1.3.

3. Data parallel version (DPV): mutliversion parallelism is exploited as in the PMV, and additionally, hierarchical refinement and resulting data parallelism is exploited for actor U, as described in Section 2.4.1.3.

Our simulations are organized into four groups. In group 1, three different hardware designs are simulated.

1. case1(a): sequential system with actor S_1;
2. case1(b): sequential system with actor S_2;
3. case2: parallel, multi-version system using both S_1 and S_2.

In Table 2.2, different values $(0, 3, 6, 9, 12)$ for the parameter $ncond$ determine different objective functions that are input to the Newton KKT system. Here $negeig$ is set to 0. For each $ncond$, we simulation the execution of the $< P_0 >$ for 50 times. For each execution, Q, c, A and b are randomly generated as described in 2.3.3, with $n = 100$. From the simulation results, the parallel, multi-version architecture outperforms both of the sequential architectures. This is because for the same class of problems, different algorithms perform better for different parameters. For example, we solve the QP problem of $< P_0 >$, if we use the same parameter $ncond = 3$, the different algorithm (S_1 or S_2) performs differently. From results in Table 2.2, we find that case2 is either the best of case1(a) and case1(b), or is better than both case 1(a) and case 1(b). Here due to the way the first kind of benchmarks is generated, the entries of Q and c increase proportionally with the value of 10^{ncond}. For example, when $cond$ is 6, the entries of Q and c are comparable to 10^5, while A and b both have entries less than 10. The condition number of the generated matrices contributes to the computational complexity.

Next, we fix the actor S as version S_1, and use two versions of actor H (H_1 and H_2). We obtain the second group of simulations, group 2, as follows:

Table 2.2: Simulation results for group 1. The simulation is conducted 50 times to obtain mean and variance.

$ncond$	Case1(a)		Case1(b)		Case2	
statistics	mean	variance	mean	variance	mean	variance
0	0.309	0.031	0.054	0.000	0.055	0.000
3	0.758	0.013	1.452	0.005	0.758	0.013
6	0.839	0.015	0.530	0.002	0.525	0.003
9	0.769	0.014	0.330	0.003	0.330	0.002
12	1.173	0.018	0.334	0.012	0.334	0.010

Table 2.3: Simulation results for group 2. The simulation is conducted 50 times to obtain mean and variance.

$ncond$	Case3(a)		Case3(b)		Case4	
statistics	mean	variance	mean	variance	mean	variance
0	1.778	0.007	2.031	0.010	1.714	0.007
3	1.875	0.003	2.040	0.005	1.859	0.003
6	1.869	0.002	1.967	0.006	1.809	0.003
9	2.834	0.001	2.888	0.012	2.803	0.012
12	3.422	0.001	3.433	0.016	3.260	0.011

1. case3(a): sequential system with actor H_1;
2. case3(b): sequential system with actor H_2;
3. case4: parallel, multi-version system using both H_1 and H_2.

In this case, $negeig > 0$ because H only activates when $negeig$ is positive. We set $negeig = 10$. The simulated results in Table 2.3 share similar properties with the results of group 1 — the multi-version architecture again provides significant performance improvement, that is, the mean time required for case4 is smaller than that for either case3(a) or case 3(b). Note that the variance is always greater than or equal to that obtained using H_1 only.

Table 2.4: Simulation results for group 3. The simulation is conducted 50 times to obtain mean and variance.

$ncond$	Case4		Case5		Case6	
statistics	mean	variance	mean	variance	mean	variance
0	2.611	0.011	0.164	0.000	0.128	0.001
3	2.175	0.004	0.541	0.002	0.532	0.000
6	3.275	0.012	0.715	0.005	0.713	0.002
9	2.784	0.008	0.766	0.003	0.756	0.006
12	4.466	0.010	1.117	0.003	1.084	0.008

Next, in group 3, we examine the effect of combining both of the instances of multi-version parallelism that we are experimenting with.

1. case4: parallel system with multi-version implementation of H (H_1 and H_2), and single-version implementation of S (S_1 only).
2. case5: parallel system with multi-version implementation of S (S_1 and S_2), and single-version implementation of H (H_1 only).
3. case6: parallel system with multi-version implementations of both H and S.

We set $negeig = 10$, and simulate the cases with different $ncond$ parameters. The simulation results for group 3 are shown in Table 2.4. From the results, it is obvious the mean time using case6 is much smaller than both case4 and both case5. It is also noticeable that the variance is generally small, although slightly larger than the best of non-parallel schemes. It is demonstrated that the more multi-version parallelism we utilize, the better the system performance.

The simulations in groups 1-3 involve functional parallelism that is achieved through the multi-version transformation. Next, we examine the effect of data parallelism. We fix

Table 2.5: Simulation results for group 4. the simulation is conducted 50 times to obtain mean and variance.

$ncond$	Case7		Case8	
statistics	mean	variance	mean	variance
0	0.794	0.009	0.725	0.008
3	1.111	0.010	1.067	0.012
6	1.206	0.012	1.110	0.013
9	0.991	0.015	0.904	0.015
12	1.633	0.018	1.531	0.016

the value of $negeig$ at 0 , and vary the $ncond$ parameter. The actor S is fixed as S_1. Under this condition, all of the problems are convex and neither H_1 nor H_2 is needed. We summarize the fourth group of experiments as follows.

1. case7: sequential system with the original version of U;
2. case8: parallel system using the data parallel version of U described in Section 2.4.1.3.

The simulation results for group 4 are shown in Table 2.5. In these results data parallelism gives improvement in system performance as defined by the mean computation time. The improvement is not so significant since the computation time of actor U itself is not dominant over the execution time of the whole system.

The results in groups 1-4 help to understand the impact of individual dataflow graph transformations in isolation. In our next group of experiments, we show the impact of applying all of the transformations together. The results in Table 2.6 compare the performance of a sequential implementation with that of a "fully-transformed" implementation — that is, and implementation that includes multi-version implementations of both H and S , as well as a data parallel implementation of U. We set $negeig = 10$, and simulate

Table 2.6: Simulation results for group 5. he simulation is conducted 50 times to obtain mean and variance.

$ncond$	Case1(a)		Case9		Percentage of improvement	
statistics	mean	variance	mean	variance	in mean	in variance
0	2.735	0.008	0.161	0.000	94.1%	100%
3	2.310	0.012	0.517	0.001	77.6%	91.6%
6	3.040	0.007	0.636	0.000	79.1%	100%
9	2.900	0.007	0.816	0.005	81.6%	28.6%
12	4.363	0.016	1.061	0.002	75.7%	87.5%

the cases with different $ncond$ parameters. Simulation results in Table 2.6 shows a large percentage of improvement not only in mean time, but also in variance.

2.4.2 Active Set Method

In MPC, the controller is often designed to solve a quadratic programming (QP) problem of $< P_0 >$, which is used to find the optimized input. The objective function is minimized at each sampling interval, in order to solve for an optimal open loop control trajectory over that horizon. Either an active set method or an interior point method is a good candidate to solve the QP problem. They each have advantages in different situations. Interior point methods usually require fewer iterations than the active set methods. But a larger number of variables is necessary in order to solve by active set methods. For either interior point or active set methods, intensive computations are required. For both methods, an excessive number of iterations is sometimes possible. Many variables and a large number of iterations result in computational complexity, and in turn result in delay of system execution. For real time applications, if the delay is bigger than the threshold of the sampling interval, MPC is not feasible.

2.4.2.1 Dataflow Model of Active Set Method

Generally, QP problems by either a Newton-KKT or an active set method. Interior point methods usually require fewer iterations than the active set methods. But a larger number of variables is necessary in order to solve by active set methods [39]. The active set method used in our work is from MATLAB. The M-file *quadprog.m* is designed for quadratic programming (QP), and the private function *qpsub.m* is especially to solve QP problems using an active set method. The algorithm is similar to the description in [40]. We do not claim that *qpsub.m* is a particularly good algorithm. The intent is primarily to demonstrate and test the RCDF approach to improving the implementation of an active set algorithm. Finding the best active set algorithm is a separate issue.

First of all, a feasible starting point is computed by either a Phase I approach or the big M method. Since computation of a feasible point is done only once for a QP problem, the burden from this computation usually is not heavy.

In the kth iteration, inequality constraints are partitioned into two sets: active (or sufficiently close to be deemed active for this iteration) and inactive. The active set for one iteration is sometimes called the working set. We define the working set W_k at iteration k for current state variable x_k as:

$$W_k(x_k) = \{i \in I \bigcup J : a_i^T x_k = b_i\}.$$

Given an iteration x_k and the current active set, we check whether x_k minimizes the objective function in the subspace defined by the active set. If not, then we compute the search direction s_k starting from the current x_k. Next, it is required to decide how

far to move along this direction. We update the current value of x_k using $x_k = x_k + \alpha_k s_k$. If $\alpha_k = 1$, there are no new constraints active at $x_k + \alpha_k s_k$, and there are no blocking constraints. Otherwise, if $\alpha_k < 1$, that is, the step along s_k was blocked by some constraints not in W_k, then a new working set W_{k+1} is constructed by adding one of the blocking constraints to W_k.

The original system model of the active set method is described in an RCDF model, as shown in Figure 2.9. Here, to simplify the notation, we define the abbreviation of *METP Edges* as *MTPE*, and *METC Edges* as *MTCE*. The functionality of actors is described as follows:

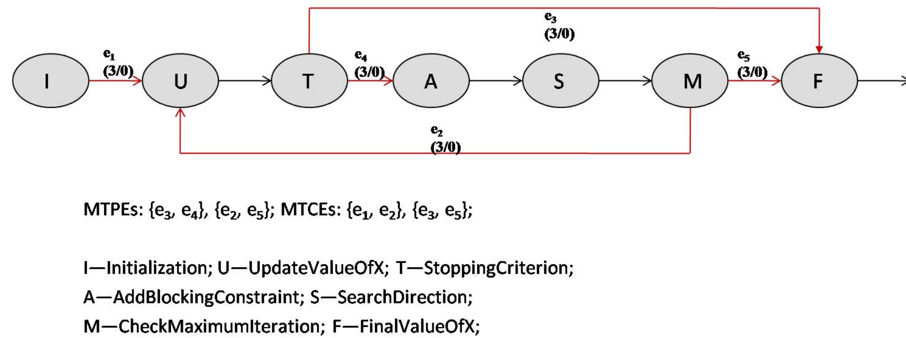


Figure 2.9: The original RCDF model of the active set method. There are two sets of MTPEs, and two sets of MTCEs. Here actor M is to check the number of iterations has not exceeded the maximum allowed.

I—The actor I is used to initialize the values of state variables and the values of parameters, such as the tolerance threshold, that are used later. The initial point x_0 in the problem P is either set manually, or computed in some way to make sure the point is feasible. In this chapter, we use a Phase I approach to compute a feasible initial point.

U—The actor U is used to compute the updated values of x. In the beginning of an iteration, the token is passed from actor I, and then the token is passed from actor M,

depending on stopping criterion and iteration number.

T—The actor T is used to check if a minimum is reached in the current subspace. Here the stopping criterion is also checked.

A—The actor A is used to add blocking constraints to the working set. As we said earlier, if the step length was blocked by some constraints not in the current active set, we add one of the blocking constraints to construct a new working set. If there is no chance of cycling then we pick the constraint which causes the biggest reduction in the cost function,i.e the constraint with the most negative Lagrange multiplier.

S—The actor S is used to compute the search direction for the next iteration. It does this by solving a linear system of equations.

M—The actor M is used to check the number of iterations. If the maximum is reached, the loop is terminated.

F—The actor F is used to record the optimized value of x.

Based on our dataflow-based modeling approach, along with MATLAB implementations of the individual actors, we have conducted MATLAB simulations to evaluate the contribution of each actor to the overall execution time required for the application.

2.4.2.2 Analysis and Improvement

Profiling

In our MATLAB simulations, the matrix Q was generated based on the condition number ($ncond$), and the number of negative eigenvalues ($ngeig$). We fixed the parameter $ngeig$ to be 0, and chose $ncond$ from the set $\{3, 6, 9\}$. It is noted that before profiling we

Table 2.7: The execution time in seconds for different actors in the active set method.

$ncond$	3		6		9	
statistics	mean	variance	mean	variance	mean	variance
I	0.003	3e-5	0.001	2e-6	0.001	4e-6
U	0.016	5e-6	0.002	1e-5	0.002	7e-6
T	0.031	9e-6	0.005	2e-5	0.004	5e-6
A	0.141	3e-5	0.125	3e-5	0.203	5e-5
S	0.078	2e-5	0.088	2e-5	0.047	1e-5

have ignored certain "fine-grained" actors that have very low computational cost. For example, we have ignored actor M, which is used only to check if the iteration number has exceeded the maximum. We have also ignored the execution time contributions of actor F, which involves a simple operation to record the final optimized value.

Table 2.1 shows the execution times of different actors for different values of $ncond$ for the first kind of benchmark problems. The units for the numbers in Table 2.7 are seconds. These values were determined by implementing each actor in MATLAB (version 7.04), executing the code for each 100 by 100 matrix Q, and recording the mean and variance of the time required to complete the computations.

According to the statistical data in Table 2.1, the computational burden from the listed actors can be ordered as follows:

$$b_A > b_S > b_T > b_U > b_I,$$

where b_{Actor} indicates the computational burden from the actor. The bottlenecks are the actors with relatively larger computational burden, and in this system, we address the actors A and S.

In the next section, we describe how our dataflow-based model together with its profiled execution time information can be used to strategically dedicate parallel hardware resources and accelerate the computation of performance-critical components in the overall design.

Improvement from Data Parallelism

Since the algorithm is decomposed into actors based on functionality, the code size and complexity generally varies across the actors. This is typical of dataflow-based program representations. Some of the more complex actors, most notably A, represent *hierarchical actors* whose internal functionalities are described by additional ("nested") dataflow graphs. We elaborate on the internal representation of actor A below.

To alleviate the bottleneck due to A, we examined the MATLAB source code for A and replaced the actor A in Figure 2.9 with the equivalent, hierarchical ("nested") RCDF subgraph shown in Figure 2.10. The detailed description of each actor is as follows:

- A_1: CheckIfHitConstraints
- A_2: CheckIfSimplexIteration
- A_3: ComputeSize;
- A_4: ComputeValueOfZ
- A_5: CheckNumberOfVariables
- A_6: UpdateFlagSignal
- A_7: ResetParameterValue

- A_8: FindDataIndexInRange
- A_9: CheckValueOfLength
- A_{10}: UpdateParametersValue
- A_{11}: DeleteColumnFromMatrix
- A_{12}: InsertColumnIntoMatrix
- A_{13}: ComputerValueOfLambda
- A_{14}: UpdateValueOfACTIND

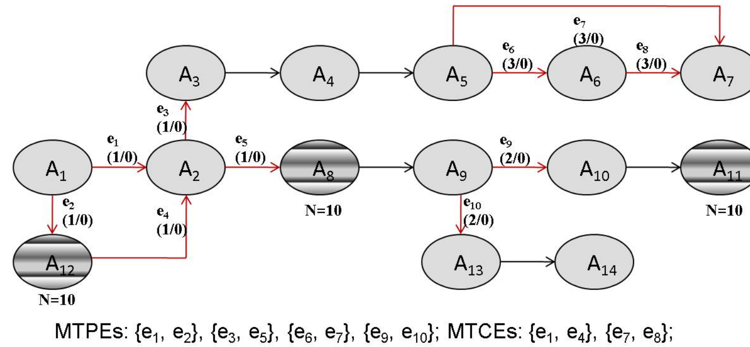


Figure 2.10: RCDF model of super actor A. Actor A is expanded into a RCDF graph.

Data parallelism exposes concurrency across sets of data to which the same operation is to be applied. Suppose there is a group of tokens indicated as $t_1, t_2, \ldots, t_n$ and there is a computational task C that can be applied to each token independently. Here, "independently" means execution of C on any given $t_i, 1 \leq i \leq n$ neither depends on nor affects any other token $t_j, j \neq i$. In this case, if we apply C to each token $t_1, t_2, \ldots, t_n$ separately at the same time, then the total execution time for processing these tokens is re-

duced by a factor of n compared to the time required by sequential processing (assuming that each token requires the same amount of processing time by C).

Exploiting data parallelism in this way can be viewed as a transformation on the given dataflow graph. We refer to this transformation as the *data parallelism transformation* (DPT).

In terms of hardware implementation, data parallel operation (the DPT) requires the use of parallel processing units. Since the computation for each token is independent, the synchronization overhead among the processing units is minimal. The fastest case arises when the number of parallel processing units is large enough so that each token can be mapped to a separate processing unit. In case the number of processing units is limited, windows of tokens are dispatched for parallel processing, and any unprocessed tokens outside the window are queued for later dispatching.

In the system shown in Figure 2.10, we can identify the actors A_8, A_{11} and A_{12} as candidates for exploiting data parallel operation. These actors are indicated as shaded actors in Figure 2.10. The original MATLAB code does not take advantage of data parallelism inside any of them. By windowing across successive groups of tokens, it is possible to apply the DPT to all of these actors. Inside each actor A_8, A_{11} and A_{12}, the original system deals with all tokens in a sequential way, as shown in the left side of Figure 2.11. After applying the DPT to modify the implementation, parallel processing units conduct concurrent groups of independent computations, as shown in the right side of Figure 2.11. For one actor, we define the *DPT degree* as the number of tokens that are processed in parallel. In Figure 2.11, the DPT degree is 3.

We simulate the active set method on the first kind of problems using a dell desktop

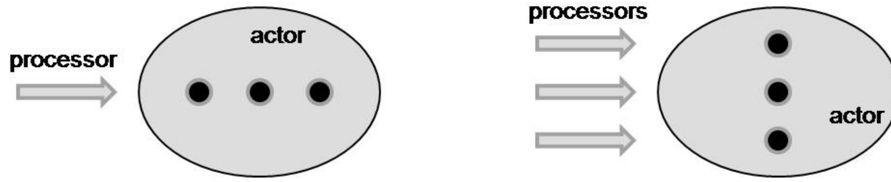


Figure 2.11: Example of execution before and after the application of DPT. On the left side, before applying DPT, single processor deals with all tokens in a sequential way; On the right side, after applying DPT, parallel processors conduct concurrent computation on a groups of tokens. The number of tokens, indicated as *degree*, is 3.

Table 2.8: Simulation results of the system before and after applying DPT.

$ncond$	system before DPT		system after DPT	
statistics	mean	variance	mean	variance
0	0.36406	0.001	0.35156	0.000
3	0.40156	0.03	0.40080	0.001
6	0.39531	0.001	0.38935	0.000
9	0.43906	0.007	0.41375	0.001
12	0.42031	0.001	0.40563	0.000

with Intel Pentium 4 CPU at 3.0GHz. We set $negeig = 10$, and simulate the cases with different $ncond$ parameters. The performance improvement due to our application of the DPT is shown as Table 2.8. In this experiment, we have applied the DPT to the following actors: A_8, A_{11}, A_{12}, and for each actor, we have used $N = 10$ as the *DPT degree*. Simulate results show a small improvement from DPT, since computation time for actor A is just a portion of the execution time of the whole system.

Improvement from Multi-version Parallelism

Besides data parallelism, the other important form of parallelism exposed effectively by dataflow graphs is functional parallelism. Functional parallelism refers to the simultaneous execution of distinct actors on separate hardware resources, whereas data

parallelism involves simultaneously executing the same actor on separate resources.

The hybrid form of parallelism, which we call *multi-version parallelism*(MVP), can be very useful when hardware resources are relatively abundant and constraints on performance are relatively stringent. We view multi-version parallelism as a hybrid form because it relates to aspects of both functional and data parallelism.

Next we solve the bottleneck due to actor S by taking advantage of functional parallelism. In the simulation, we use two different search algorithms for determining the search direction: Newton's method denoted by actor S_1 and steepest decent denoted by actor S_2.

The time required to complete an execution of S_1 or S_2 is in general data-dependent, and the relative speeds of corresponding executions (i.e., executions that have the same "j" index) are also data-dependent. In general, for some values $j \in J_1$, each jth execution of S_1 will complete before the jth execution of S_2, and for other values $j \in J_2$ ($J_1 \cap J_2 = \emptyset$), the jth executions of S_2 will complete sooner.

A multi-version implementation of the search direction calculation based on alternative implementations S_1 and S_2 therefore involves executing them both in parallel (simultaneously on separate resources), and taking the result of the S_i that finishes first. As soon as one of the "versions" completes, its result is taken as the result of the corresponding execution of the search direction calculation, and the current execution of the other version is terminated. Such a multi-version implementation is useful whenever there can be significant variation between which of the versions completes first, and the available hardware resources accommodate parallel execution of the different versions.

The modified system with MVP is shown in Figure 2.12. Actor R in Figure 2.12

Table 2.9: Simulation Results of System with and without MVP.

$ncond$	system with S_1		system with S_2		MVP system	
statistics	mean	variance	mean	variance	mean	variance
0	0.350	0.000	0.252	0.001	0.252	0.000
3	0.402	0.002	0.450	0.004	0.401	0.001
6	0.381	0.001	0.323	0.001	0.323	0.001
9	0.438	0.013	0.428	0.001	0.428	0.000
12	0.402	0.008	0.455	0.002	0.402	0.008

represents a special actor called as *multi-version output selector* (*MVOS*) [25]. The details of MVOS and *self loop* edge of R in Figure 2.12 is of the same function as in Figure 2.7.

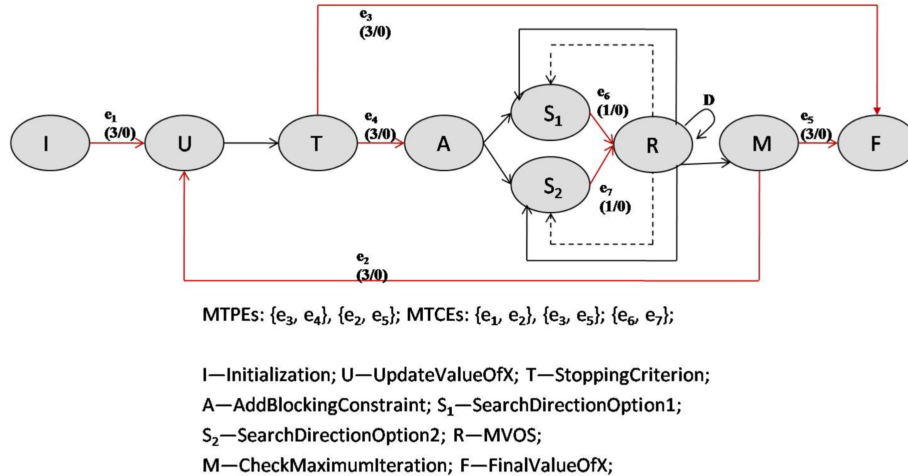


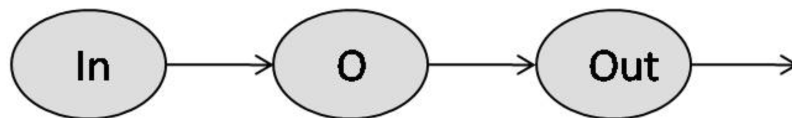
Figure 2.12: RCDF model of active set method after application of multi-version transformations. This is parallel version of active set method implementation.

We apply MVP to the system with DPT. The parallel version of actor S is used in the system. We set $negeig = 10$, and simulate the first kind of benchmarks with different $ncond$ parameters. The system performance improvement is shown as Table 2.9. Besides improving system performance, MVP also plays an important role when handling failed operations, which is critical to real-time system.

2.4.3 Hierarchical System Model

RCDF specifications, as with most other forms of dataflow specifications, can be constructed hierarchically. A hierarchical RCDF graph can contain one or more *hierarchical actors* (also called super actors), where a hierarchical actor is an actor whose internal functionality is represented by a (nested) RCDF graph. Such a nested RCDF graph can also be hierarchical, so hierarchical RCDF specifications can be arbitrarily "deep." By applying hierarchical organization to Figure 2.9, we can derive the hierarchical RCDF graph shown in Figure 2.13. The actor O in the hierarchical graph represents the nested graph of Figure 2.9.

We apply MVP in the hierarchical graph of Figure 2.13, and the graph that results from this combination of hierarchy and MVP is shown in Figure 2.14. The performance improvement associated with this transformed representation is shown in Table 2.10. In these results, the mean time of computation always decreased, and the variance is substantially decreased on all but one case. It demonstrated that the system performance can be improved if we increase the size of hierarchy with introducing more levels of parallelism.



In—InputFunction; O—SolveOptimizationProblem;
Out—OutputResult;

Figure 2.13: Hierarchical dataflow representation. This is top-level graph, where actor O is a super actor, which can be expanded into a subgraph.

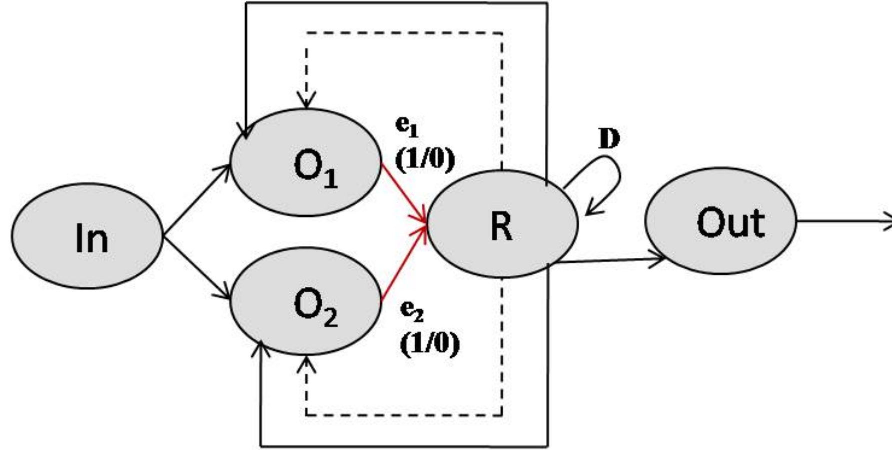


Figure 2.14: MVP version of hierarchical RCDF model. MVP is applied to actor O. The original actor O in Figure 2.13 is transformed into parallel version.

2.5 Improved Linear System solvers

From Table 2.1, we can see S is one of the computational bottlenecks. The main computational part of S is to solve a linear system of equations. Similar linear systems of equations play an important role in many problems in control and signal processing, including in both the Newton-KKT and active set algorithm. Because of the great impor-

Table 2.10: Simulation results of the system in higher hierarchy.

$ncond$	system without MVP		system with MVP	
statistics	mean	variance	mean	variance
0	0.29844	0.010	0.15937	0.000
3	0.29375	0.007	0.29370	0.005
6	0.36594	0.003	0.36500	0.002
9	0.50313	0.008	0.36406	0.003
12	0.53281	0.005	0.28594	0.007

tance of solving linear equations in science and engineering there is a vast literature on the parallel computation of the solution to such problems. This literature is both complex and confusing because parallel computing can be very sensitive to the details of the computer architecture as well as to the algorithm used. Furthermore, because solving such problems is one of the benchmarks for determining the fastest computer, programmers have considerable incentive to develop special tricks to make specific computers solve such problems quickly.

However, it is clear from the literature that very large improvement in the speed with which linear equations are solved is possible using various forms of parallel computing. Furthermore, there is a large variety of parallel hardware and this collection is rapidly increasing. In order to take advantage of this we have first enhanced RCDF to include a way to account for communication delays because such delays are very significant in highly parallel computing. We have also begun to explore ways to implement large amounts of parallelism in the linear equations solver that is a major component of both Newton-KKT and active set methods for solving QPs. This work is described below.

2.5.1 Gaussian Elimination

Gaussian Elimination (GE) is a general way to solve linear systems of equations and its parallel implementation has been heavily studied. Thus, although the QR method is arguably better for the class of problems of interest here, it is better to begin with GE. Implementations of parallel Gaussian Elimination depend on the parallel hardware platform, such as a multiprocessor or multi-core system. Computations in each processing unit are

similar to each other. However execution of the computations requires the collaboration of all the units.

The problem, indicated as $< P_2 >$ we wish to solve has the form

$$Ax = b \tag{2.1}$$

Note that the A and b here are completely different from the A and B in the MPC problems. Here A is simply a square invertible matrix and b is a vector of commensurate size.

If we assume, for simplicity, that the diagonal elements of the matrix A are all not zero, the critical part of a sequential program for GE is shown in Figure 2.15. In Figure 2.15, the key computations are located in 1.2.1, 1.3.1.1 and 1.3.2. These computations can be implemented in a parallel way.

```
1: for (t=0; t<(Size-1); t++) {
    1.1: pivot = a[t][t];
    1.2: for (i=(t+1); i<Size; i++) {
            1.2.1: m[i][t] = a[i][t] / pivot;
    }
    1.3: for (i=(t+1); i<Size; i++) {
            1.3.1: for (j=t; j<Size; j++) {
                    1.3.1.1: a[i][j] = a[i][j] - m[i][t] * a[t][j];
            }
            1.3.2: b[i] = b[i] - m[i][t] * b[t];
    }
}
```

Figure 2.15: Sequential Program of GE. The intensive computations are located in 1.2.1, 1.3.1.1 and 1.3.2.

Grid computation, such as in ScaLAPACK, is typical for parallel Gaussian Elimi-

nation. In this way, key computations are identified and then distributed in a processor matrix.

Figure 2.16 presents our RCDF model of one processing unit for Gaussian Elimination.

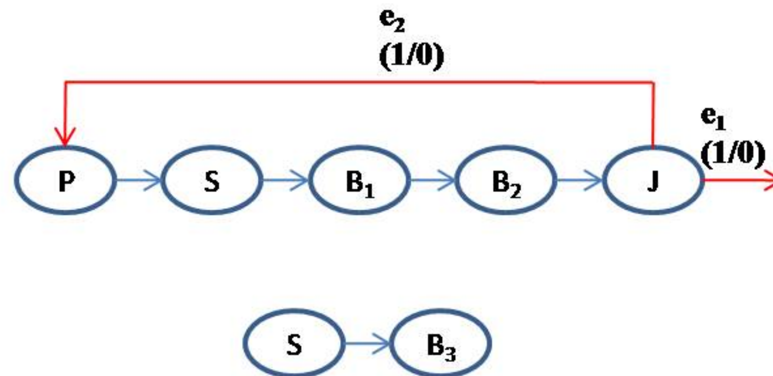


Figure 2.16: RCDF model of Gaussian Elimination on a single processing unit.

In the RCDF model above, there is a particularly important set of actors, indicated by BLASn (Basic Linear Algebra Subprograms). BLASn represents a series of fundamental linear algebra computations. They can be considered to be a library to perform basic linear algebra operations such as vector and matrix multiplication. The BLAS are used to build larger packages such as LAPACK. Because they are heavily used in high-performance computing, highly optimized implementations of the BLAS have been developed by hardware vendors such as Intel and AMD. The LINPACK benchmark relies heavily on DGEMM, a BLAS subroutine, for its performance.

Parallel Gaussian Elimination requires the collaboration of multiple processing units.

Although each processing unit conducts similar computation tasks, they have to communicate with each other to swap the data and calculate the final result. An RCDF model to implement Gaussian Elimination on 4 processors is shown in Figure 2.17. In this model, the processor matrix for GE is 2×2. Note that communication edges have been introduced to indicate the communication between two actors. This is different from other types of edges in RCDF models; there is no specific token related to communication edges.

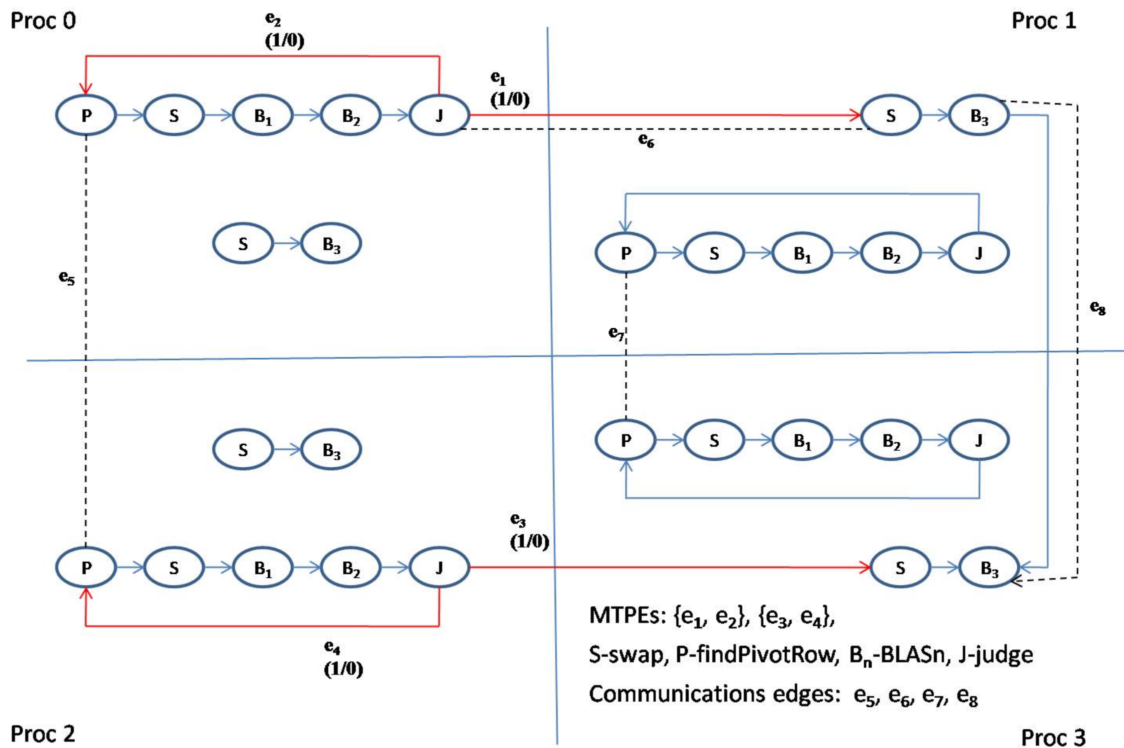


Figure 2.17: RCDF model of Gaussian Elimination on four processing units.

In our target architecture model, we map processors onto a 2-dimensional matrix in a block-cyclic distributed manner. Such an arrangement is represented in the form $n_r \times n_c$, where n_r represents the number of processors in a row, and n_c represents the number of processors in a column of the target architecture matrix. The matrix to be processed is also divided into a 2-D pattern, based on homogeneous blocks of size $m_r \times m_c$, where

$m_r \leq n_r$, and $m_c \leq n_c$. In our experiments, it is assumed that $m_r = m_c$ (i.e., each block in the pattern has a "square" arrangement). The computations related to blocks are allocated to the processor pattern in modulo fashion — after a computation is mapped to the last row or column, the mapping process "wraps around" cyclically to the first row or column, respectively. An example of mapping a set of 5×5 matrix computations onto a $2x2$ processor pattern is shown in Figure 2.18.

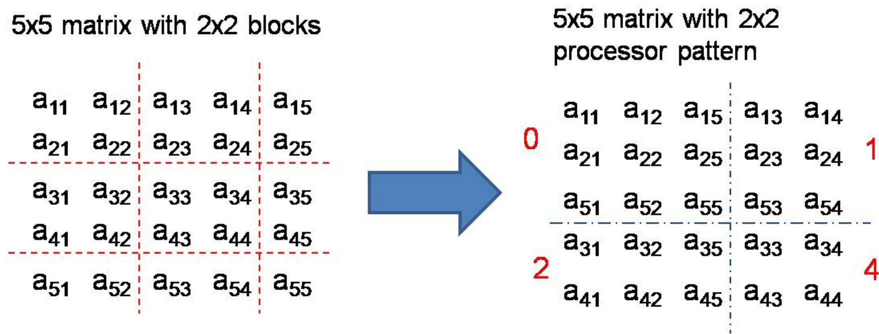


Figure 2.18: 2-D Block cyclic distribution of computations onto parallel processing units. After a computation is mapped to the last row or column, the mapping process wraps around cyclically to the first row or column, respectively.

We simulate our model of a distributed memory environment using the Message Passing Interface (MPI). MPI is commonly used to simulate the communications between different processing units in a system with distributed memory. One of the major aspects of implementing the Gaussian Elimination algorithm on a distributed memory system is that the communication time has to be taken into account when calculating the total execution time. In general, interprocessor communication time has a significant effect on the performance of algorithms on multiprocessor systems.

In our experiments, we use a constant time of 0.002sec as the communication overhead between any two processors. Whenever there is communication between two proces-

sors, as indicated by the communication edges shown in Figure 2.17, the communication overhead estimate is added onto the total execution time.

By applying the Schur complement to problem $< P_1 >$ [38], we decreased the dimension of the linear system from 1200x1200 to 400x400. We simulate the second kind of benchmark problems using a dell desktop with Intel Pentium 4 CPU at 3.0GHz. We tested the Gaussian Elimination algorithm with different processor patterns given the fixed block size to be allocated in each processor. The $PDGESV$ routine in ScaLAPACK is used in the simulation. For each processor pattern, we execute 20 times to obtain mean and variance value of the execution time. In each execution, A, B and C are randomly generated; and x_0 is a vector with each entry as 0.5. The numbers in the table were determined in the following way. The benchmark QPs set up earlier, see $< p_0 >$, involving $\hat{A}$, $\hat{C}$, $\hat{d}$ with $\hat{u}$ as the unknown to be computed were input to our improved implementation of Newton-KKT. This created a large system of linear equations to solve. This system has some structure which we exploited to simplify the computations slightly. Almost all of this special structure is always present in QPs derived from an MPC problem. We then applied parallel GE and the simulation results are shown in Table 2.11.

The simulation results reflect the effect of communication time between processors. The system performance does improve with an increasing number of processors, however, the rate of increase decreases as the number of processors used in the computation increases. The reason is that it takes time for the processors to communicate and synchronize with each other. The portion of communication time in the total execution time increases with the increasing number of processors.

In our simulation, we assume that all the processors are homogeneous, which means

Table 2.11: Simulation results for parallel Gaussian Elimination with different processor patterns.

Process pattern	mean	variance
2x2	2.000129	0.102159
2x4	1.035022	0.011020
2x8	0.795640	0.072004
2x16	0.581850	0.025809
2x2	1.985206	0.100000
4x2	1.029348	0.021832
8x2	0.808240	0.052389
16x2	0.562020	0.013480

Table 2.12: Simulation results for parallel Gaussian Elimination with different block size.

Block size	mean	variance
5	1.193409	0.032600
10	0.906433	0.052802
20	0.795640	0.072004
30	0.606800	0.012560
40	0.523929	0.021430
80	0.752324	0.014398

that each processor has the same capacity of computation. Under this assumption, the processor pattern 2×4 results in the same speed as the pattern of 4×2. The results will change if we apply heterogeneous processors.

We also tested the parallel Gaussian Elimination algorithm with the same processor pattern but different block sizes. The same matrices were used as in the tests that determined the values in Table 2.11. The simulation results are shown in Table 2.12.

Simulation results indicate that the mean computation achieves its peak when the block size is 40. The variance is more complicated as it is smaller for both the 30 and 80 block size than for the 40 block size. Nonetheless, the 40 block size is clearly best. This

Table 2.13: Simulation results for MPC problems with parallel Gaussian Elimination.

simulation setting	include simulation time		exclude simulation time	
statistics	mean	variance	mean	variance
seq	10.3689	1.309000	10.3689	1.309000
newton-KKT	6.27500	0.004994	6.27500	0.004994
pges	7.36600	0.247642	7.00418	0.182920
pgen	3.92840	0.024239	3.66000	0.021875

is the effect of communications between different processors. In the extreme case, if we allocate the computation of each entry of the matrix into its own processor, the communication time will dominate the execution time. On the contrary, if we allocate the whole matrix as one block, it turns out to be a sequential Gaussian Elimination instead of a parallel version. If we integrate the parallel Gaussian Elimination with the 2×8 pattern into the Newton-KKT system, the simulation results are shown in Table 2.13. In Table 2.13, seq denotes a sequential implementation of MPC problems using the standard Newton-KKT method; $newton - KKT_s$ denotes parallel implementation without a parallel linear solver; $pges$ is sequential implementation with only the linear system solver executed in a parallel way; $pgen$ is a parallel implementation with parallel Gaussian Elimination. Simulation results show that the parallel implementation of $pgen$ requires less than 40% of the time for sequential version seq and has greatly reduced the variance. We also simulate the extreme case that the communication time between processors is ignored an the mean execution time is further improved. When the system is implemented in parallel processors, communication time is one of the factors which affect the system performance and it is highly hardware-dependent.

The performance of Newton-KKT with parallel Gaussian elimination will be fur-

ther improved if we use more processors than 16. However, with 2×8 processors, the computation time on linear systems counts as 20% in $pgen$ version. If the whole system performance already meets the real time requirement, or there are other actors spending more execution time, we can conclude that the actor S is no longer the computational bottleneck; it is not necessary to consume more hardware resources. So introduction of parallelism is flexible when tradeoff between system performance and hardware resource management.

2.5.2 QR Decomposition

In linear algebra, the QR decomposition (also called a QR factorization) of a matrix is a decomposition of the matrix into an orthogonal and a upper triangular matrix. QR decomposition is often used to solve the linear least squares problem, and is the basis for a particular eigenvalue algorithm, the QR algorithm. Here we use it as alternative method to solve linear systems.

For the problem $< P_2 >$, we try to solve the system of the linear equation $Ax = b$ for the unknown x. The QR decomposition, $A = QR$, allows us to transform this general system into a simpler problem, that is,

$$Ax = b \Leftrightarrow QRx = b \Leftrightarrow Rx = Q^*b$$

If we assume A is $m \times n$ matrix, then Q is $m \times n$ orthogonal matrix and R is $m \times n$ real upper triangle matrix. Q^* is the transpose matrix of Q. If A is nonsingular, this factorization is unique.

There are several methods for actually computing the QR decomposition, such as by means of the Gram Schmidt process, Householder transformations, or Givens rotations. Each has a number of advantages and disadvantages. Householder transformation is used in our work. The algorithm consists of applying successive Householder transformations to the matrix A. After Q and R are determined, we can address the simpler problem of solving a linear system of the form $Rx = c$, where R is upper triangular, i.e., $r_{jk} = 0$ if $j > k$, and $c = Q^*b$. The procedure of back substitution is applied to solve the simpler linear system. This method is simpler than applying Gaussian elimination to a general (non-triangular) linear system in the sense that all the hard work was accomplished when we computed the QR factorization.

The RCDF model of the QR decomposition in one processing unit is shown in Figure 2.19. The detailed description of each actor is as follows:

- Q: CalculateHouseholderMatrix
- T: Transformation
- J: JudgeIfMeetPattern
- P: PanalFactorization (Super Actor)

We use ScaLAPACK to implement a parallel version of QR decomposition. To achieve high performance on modern computers with different levels of cache, the application of the Householder transformation is implemented block wise as shown in Figure 2.18. Assume k is block size, k elementary Householder matrices are accumulated within a panel (a block-column) V consisting of k reflectors. The consecutive applica-

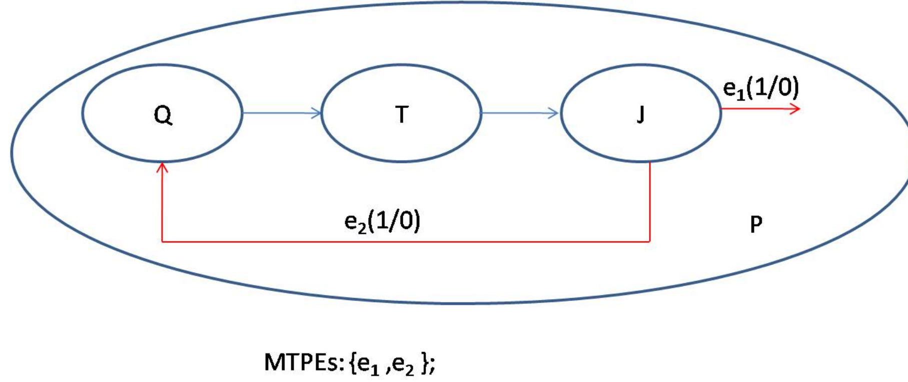


Figure 2.19: RCDF model of Panal Factorization in one processing unit. P is a super actor which can be expanded into a subgraph including Q, T and J.

tions of these k reflectors ($H_1 H_1 ... H_k$) is then constructed all at once using the matrix equality $H_1 H_2 ... H_k = I - VTV_T$ (T is a $k \times k$ upper triangular matrix). However, this blocking incurs an additional computational overhead. The overhead is negligible when there is a large number of columns to be updated but is significant when there are only a few columns to be updated. The RCDF model of QR decomposition on four processing units is shown in Figure 2.20. Compared with Gaussian Elimination model, there are fewer communication edges between processing units.

In our experiments, we use a constant time of 0.002sec as the communication overhead between any two processors. Whenever there is communication between two processors, as indicated by the communication edges shown in Figure 2.20, the communication overhead estimate is added onto the total execution time.

By applying the Schur complement to problem $< P_1 >$ [38], we decreased the dimension of the linear system from 1200x1200 to 400x400. We simulate the second kind of benchmark problems using a dell desktop with Intel Pentium 4 CPU at 3.0GHz. We tested the QR decomposition algorithm as linear system solver with different processor

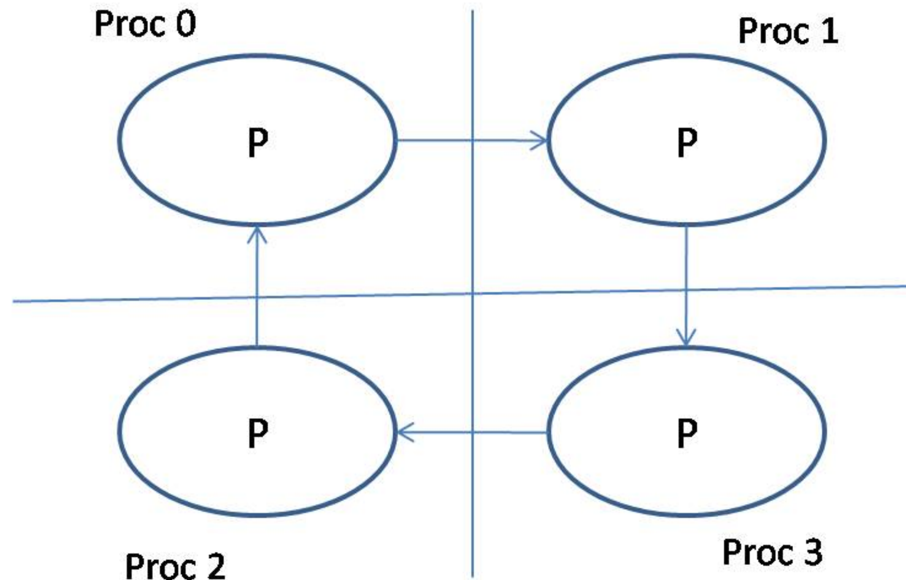


Figure 2.20: RCDF model of Panal Factorization on four processing units. There are four processing units. P is a super actor which can be expanded into subgraph as in Figure 2.19.

patterns given the fixed block size to be allocated in each processor. For each processor pattern, we execute 20 times to obtain mean and variance value of the execution time. In each execution, A, B and C are randomly generated; and x_0 is a vector with each entry as 0.5. The PDGEQRF routine in ScaLAPACK is used in the simulation and the PDGEQR2 routine is used to perform the panel factorizations. The simulation results are shown in Table 2.14.

The simulation results indicate the effect of communication time between processors. The system performance does improve with an increasing number of processors, however, the rate of increase decreases as the number of processors used in the computation increases. The reason is that it takes time for the processors to communicate and synchronize with each other. The portion of communication time in the total execution

Table 2.14: Simulation results for parallel QR decomposition with different processor patterns.

Process pattern	mean	variance
2x2	1.500000	0.092023
2x4	0.732038	0.019823
2x8	0.608120	0.060000
2x16	0.380194	0.019813
2x2	1.520100	0.002203
4x2	0.700241	0.010839
8x2	0.601268	0.051431
16x2	0.390005	0.073181

time increases with the increasing number of processors.

In our simulation, we assume that all the processors are homogeneous, which means that each processor has the same capacity of computation. Under this assumption, the processor pattern 2×4 results in the same speed as the pattern of 4×2. The results will change if we apply heterogeneous processors.

We compare the performance of both Gaussian Linear system solver and QR linear system solver, as shown in Figure 2.21.

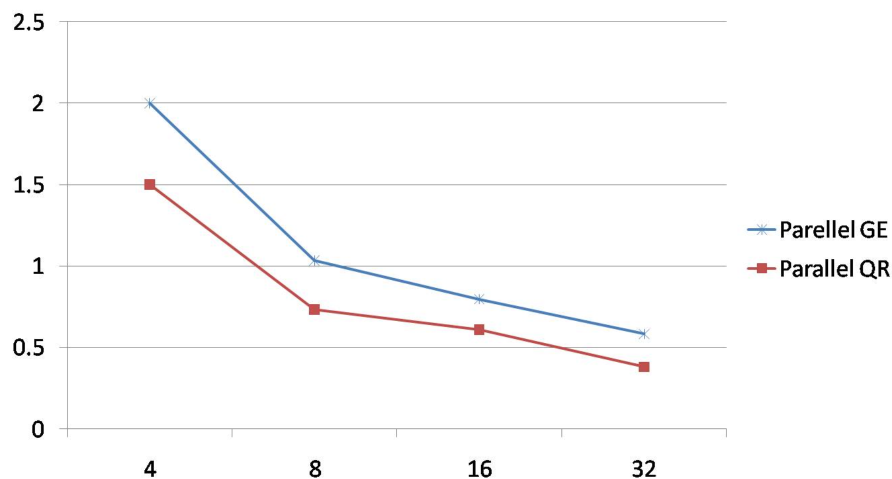


Figure 2.21: Performance comparison of parallel GE and QR.

Table 2.15: Simulation results for MPC problems with parallel QR decomposition.

simulation setting	include simulation time		exclude simulation time	
statistics	mean	variance	mean	variance
seq	10.3689	1.309000	10.3689	1.309000
newton-KKT-s	6.27500	0.004994	6.27500	0.004994
pges-QR	5.81930	0.152302	5.56042	0.121873
pgen-QR	3.08820	0.019210	2.87952	0.010528

If we integrate the parallel QR decomposition with the 2×8 pattern into the Newton-KKT system, the simulation results are shown in Table 2.15. In Table 2.15, seq denotes a sequential implementation of MPC problems using the standard Newton-KKT method; $newton - KKT - s$ denotes parallel implementation without a parallel linear solver; $pges - QR$ is sequential implementation with only the linear system solver executed in a parallel way; $pgen - QR$ is a parallel implementation of Newton-KKT with parallel QR decomposition. Simulation results show that the parallel implementation of $pgen - QR$ requires less than 30% of the time for sequential version seq and has greatly reduced the variance. We also simulate the extreme case that the communication time between processors is ignored an the mean execution time is further improved. When the system is implemented in parallel processors, communication time is one of the factors which affect the system performance and it is highly hardware-dependent.

The performance of Newton-KKT with parallel QR decomposition will be further improved if we use more processors than 16.

2.6 Application and Simulation

We take an example due to Maciejowski et al. where MPC is used to control a Cessna citation 500 aircraft [41] when it is cruising at an altitude of 5000m and a speed of 128.2 m/sec. The structure of the implementation of the whole system is shown in Fig. 2.22. As shown in Fig. 2.22, the active set method described above is used to produce the control input.

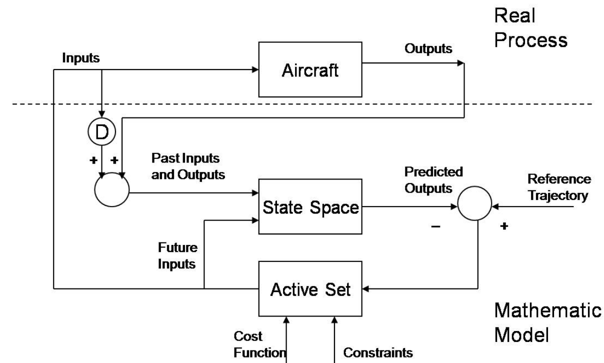


Figure 2.22: Complete MPC system of aircraft.

There is only one input: elevator angle. There are three outputs: pitch angle, altitude and altitude rate.

The following is the constant-speed approximation of linearized dynamics of the aircraft:

$$\dot{x} = Ax + Bu,$$

$$y = Cx,$$

where

$$A = \begin{bmatrix} -1.2822 & 0 & 0.98 & 0 \\ 0 & 0 & 1 & 0 \\ -5.4293 & 0 & -1.8366 & 0 \\ -128.2 & 128.2 & 0 & 0 \end{bmatrix}, B = \begin{bmatrix} -0.3 \\ 0 \\ -17 \\ 0 \end{bmatrix},$$

and

$$C = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -128.2 & 128.2 & 0 & 0 \end{bmatrix}, D = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

The elevator angle is limited to $\pm 15^o (\pm 0.262 rad)$, and the elevator slew rate is limited to $\pm 30^o/sec$. These are limits imposed by the equipment design, and cannot be exceeded. For passenger comfort the pitch angle is limited to $\pm 20^o$.

The performance measure for this application is formulated as a QP problem: $f(z) = \frac{1}{2} z^T Q z + c^T z$, and $Jz \leq g$. An MPC controller was designed by Maciejowski et al. with the sampling interval $T_s = 0.5s$, the prediction horizon $N_p = 10$, the control horizon $N_u = 3$. The following constraints : $|u| \leq 0.262$, $|\Delta u| \leq 0.524$, $|y_1| \leq 0.349$ were also included. The system has to solve on-line the QP problem at every sampling time.

Table. 2.16 shows the time required on the average for one time step assuming each actor is implemented in MATLAB and that parallel actors run simultaneously. The second row of Table. 2.16 indicates what would happen for a much smaller sampling interval. For this problem, such a short sampling interval would be unwise because it results in a relatively poorly conditioned problem. Even so, if the algorithm using PMV+DPT were programmed into an FPGA, for example, it would run quickly enough to be usable with

Table 2.16: Average computing time in seconds for aircraft example.

$T_s(sec)$	original	MVP+DPT	improve
0.5	0.10643	0.04039	62.0%
0.05	0.10724	0.04872	54.5%

the faster dynamics and it is not clear that the original implementation could be made fast enough to be usable.

Another application is to the simplified paper machine model from [42]:

$$x(k+1) = 0.6x(k) + Bu(k),$$

$$y(k) = x(k) + d(k);$$

where

$$B = \begin{bmatrix} 1 & 0.7 & 0.4 & 0.1 & 0 & 0 & \dots & 0 \\ 0.7 & 1 & 0.7 & 0.4 & 0.1 & 0 & \dots & 0 \\ \vdots & & & & & & & \vdots \\ 0 & \dots & 0 & 0 & 0.1 & 0.4 & 0.7 & 1 \end{bmatrix} \in R^{n \times n}$$

and $x(k) \in R^{n \times n}$, $y(k) \in R^{n \times n}$, $u(k) \in R^{n \times n}$. The disturbance vector $d(k)$ is assumed to be -1, $\forall k \geq 1$. The initial profile is $y(0) = 0$. The objective function to be minimized is:

$$\Sigma_{i=1}^{10}(r(k+i) - y(k+i \mid k))^T(r(k+i) - y(k+i \mid k)),$$

Table 2.17: Average computing time in seconds for paper machine model.

n	original	MVP+DPT	improve
20	0.30825	0.11652	62.2%
40	0.71596	0.20388	71.8%

where r is set point and assumed to be 0.63.

The objective function above can be transformed into a QP problem identical on the form to P_0. The detailed procedure is given in [42].

We approximate the system with a sampling time of 1 second. The simulation is conducted on a Dell desktop with Intel Pentium 4 CPU at 3.0GHz. Table 2.17 shows the time required on the average for one time step assuming each actor is implemented in MATLAB and that parallel actors run simultaneously.

Chapter 3

Exploiting Statically Schedulable Regions

3.1 Overview

In this chapter, we apply our proposed framework to video processing systems. We present how dataflow techniques analyze the existing system in order to obtain efficient implementation, as shown as red, bold and italic in Figure 3.1. Our work presented in this chapter is also available in the publication [43].

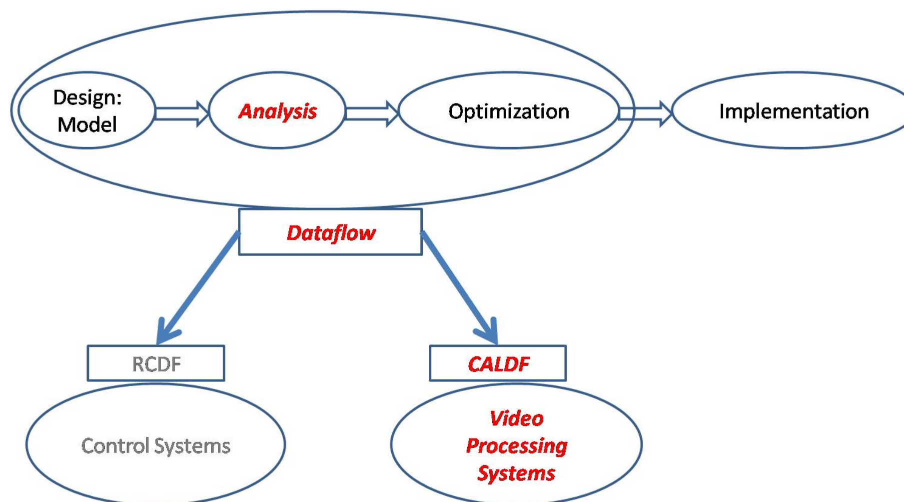


Figure 3.1: Framework for video processing systems: Analysis.

Dataflow-based programming is employed in a wide variety of commercial and

research-oriented tools related to DSP system design. Synchronous dataflow (SDF) is a specialized form of dataflow that is streamlined for efficient representation of DSP systems [14]. SDF is a restricted model that handles a limited sub-class of DSP applications, but in exchange for this limited expressive power, SDF provides increased potential for static (compile-time) optimization of DSP hardware and software (e.g., see [44]).

Since the introduction of SDF, a variety of more general dataflow models of computation have been proposed to handle broader classes of DSP applications. These alternative modeling approaches provide different trade-offs among expressive power, optimization potential, and intuitive appeal. In general, they provide enhanced expressive power, but cannot directly utilize static scheduling techniques, such as those that have been developed for SDF.

A variety of dataflow-based languages and tools have been developed for design an implementation of embedded DSP systems. For example, CAL [1] is a language for specifying dataflow actors in a way that is fully general (in terms of expressive power), while clearly exposing functional structures that are useful in detecting important special cases of actor behaviors (e.g., SDF or SDF-like actor behaviors). The CAL language, in terms of its high level of abstraction, is similar to the Stream-Based Functions (SBF) model of computation [19]. Both models share common points to describe dynamic systems, such as input/output ports in CAL and read/write ports in SBF, actions in CAL and functions in SBF, and internal states in both models. However, SBF combines the semantics of both dataflow models and process network models, while CAL extends the dataflow model by enriching the properties of single actors. In general, CAL is a fully-featured programming language, providing both an abstract, dataflow model of computation as well as a

comprehensive set of operators and other semantic features for completely specifying the internal behavior of dataflow components.

DIF [8] is a language for specifying dataflow graphs in terms of subsystems that conform to different kinds of specialized dataflow modeling techniques, and The DIF Package (TDP) is a tool for analyzing DIF language specifications, with emphasis on scheduling- and memory-management-related analysis techniques [8]. CAL2C [45, 9] is a tool that performs automatic generation of C code from CAL networks, thereby providing a direct bridge between CAL and off-the-shelf embedded processing platforms. CAL2C is now part of Open RVC CAL Compiler (Orcc). Orcc is described in [46] and can be downloaded in [47].

In this chapter, we explore an integration of CAL, TDP, and CAL2C, including the introduction of new models and analysis methods to formally link these tools. Through this linkage, we develop novel methods for *quasi-static scheduling* of dynamic dataflow graphs. Here, by quasi-static scheduling, we mean scheduling techniques in which a significant proportion of scheduling decisions are fixed at compile time — thereby promoting predictability and optimization — and integrated with a relatively small proportion of dynamic scheduling decisions, which provide for increased generality and run-time adaptability compared to fully static scheduling.

More specifically, in this chapter we introduce the concept of a *Statically Schedulable Region* (SSR) in a dataflow graph, and demonstrate the utility of this concept in quasi-static scheduling. We also propose an automated method to detect SSRs, using the TDP tool, in DSP applications that are modeled by the CAL language. The efficiency of quasi-static schedules built from SSRs is demonstrated by evaluating synthesized C-code

implementations that are generated using CAL2C.

After extracting SSRs from a dynamic CAL network, we can take advantage of existing SDF scheduling methods to schedule the different SSRs. More specifically, in this chapter, we introduce the concept of an *SSR actor*, which is a subsystem within an SSR that can be treated as an SDF actor for purposes of scheduling. In terms of the components in the original CAL specification, an SSR actor may correspond to a single CAL actor or part of (a subset of the functionality within) a CAL actor. Scheduling based on SSR actors is thus of significantly more general applicability compared to conventional SDF scheduling, where SDF actors in the original specification are treated as indivisible "black boxes".

SSRs, together with their application to static and quasi-static scheduling, benefit not only sequential implementations, but also implementations on parallel processing systems, such as multi-core processors. Along with our method for automatically deriving SSRs, we propose an SSR-based transformation technique for mapping dynamic CAL networks onto multi-core platforms. We demonstrate that our techniques result in significant improvements in system performance compared to conventional actor-based mapping approaches.

A preliminary, partial summary of this chapter was presented in [43]. This chapter incorporates the following further developments compared to the earlier presentation of [43]. First, we develop a precise and comprehensive formulation of our SSR detection methods in terms of relevant graph-theoretic concepts. Second, we discuss how capabilities of TDP can be exploited in new ways to achieve efficient scheduling of SSRs. We also discuss how integrating our methods for SSR detection and TDP-based scheduling

into CAL2C provides capabilities for efficient, automated implementation of video processing systems. Third, we explore novel techniques for mapping CAL networks into multi-core systems by grouping dynamic ports with SSRs into a form of subsystem that we call *weakly connected SSRs*. Our transformation techniques are demonstrated on the IDCT module of an MPEG Reconfigurable Video Coding (RVC) system and an MPEG-4 RVC simple profile (SP) decoder.

This chapter is organized as follows. Section 3.2 introduces previous work related to the CAL language. Section 3.3 outlines our methods and notations for translation and analysis across different modeling languages. In Section 3.4, we introduce the concept of SSRs, and develop a detailed procedure for deriving SSRs from CAL networks. Section 3.5 defines the concept of *SSR actors*, and describes how this special class of SSRs can help in exploiting existing SDF scheduling techniques and tools within a dynamic dataflow context. Simulation results on an IDCT module are also presented in this section. Section 3.6 explores methods to implement CAL networks based on the concept of weakly-connected SSRs. Simulation results on an MPEG-4 RVC SP decoder are presented in this section.

3.2 CAL and Scheduling of CAL Systems

CAL is a dataflow- and actor-oriented language that describes algorithms in terms of networks of communicating dataflow-actor components. A CAL actor is a modular component that encapsulates its own state. The state of an actor is not shareable with other actors, and thus, an actor cannot modify the state of another actor.

The behavior of an actor is defined in terms of a set of *actions*. The operations an action can perform are consumption (reading) of input tokens, modification of internal state, and production (writing) of output tokens. The topology of the connections among actor input and output ports constitutes what is called a *CAL network*. Compared to the complexity of actors, edges — connections between pairs of actors — are rather simple. The only interaction an actor can have with another actor is through input and output ports that connect the actors. Such connections are represented as edges in a CAL network.

Each action of an actor defines the kinds of transitions that internal states can undergo, and the specific conditions under which the action can be executed (*fired*). The conditions for firing actions in general involve (1) the availability of input tokens, (2) values of input tokens, (3) state of the actor, and (4) priority of the action. In an actor, actions are executed sequentially — i.e., at most one action can be executing at any given time.

CAL is supported by a portable interpreter infrastructure that can simulate a hierarchical network of actors. In addition to the strong encapsulation afforded by the actor description, the dataflow model also makes much more algorithmic parallelism explicit. This allows application of the wide range of dataflow graph transformations to the realization of signal processing systems on a variety of platforms. In particular, platforms will differ in their degree of parallelism, which gives rise to the challenging problem of matching the concurrency of the application representation with the parallelism of the computing machine that is executing it. The newly developed MPEG video coding standard, Reconfigurable Video Coding (RVC) [48], uses the CAL actor language [1] for specifying functional components, and dataflow as the composition formalism [49].

An integrated set of tools related to CAL are presented in OpenDF [50]. Among

these, we are especially interested in the available code generators that translate CAL into C or hardware description language (HDL) code.

However CAL models themselves are too general to be scheduled efficiently through any sort of direct mapping. In a direct mapping from CAL semantics, the scheduling of actor functions is resolved only at run-time, such as through the SystemC-based scheduling approach that is used in CAL2C. A number of related efforts are underway to develop efficient scheduling techniques for CAL networks. The approach of Platen and Eker [51] sketches a method to classify CAL actors into different dataflow classes for efficient scheduling. Boutellier et al. [52] propose an approach to quasi-static multiprocessor scheduling of CAL-based RVC applications. The approach involves the dynamic selection and execution of "piecewise static schedules" based on novel extensions of flow shop scheduling techniques.

Many previous research efforts have focused on task mapping for multiprocessor systems from other kinds of specification models or languages (e.g., see [16]). For example, Li et al. [53] provide a method for allocating and scheduling tasks using a hybrid combination of genetic algorithm and ant colony optimization. The approach involves consideration of both global and local memory spaces across the targeted multiprocessor system. Ennals et al. [54] develop a method for partitioning tasks on multi-core network processors.

Compared to prior work on dataflow techniques and multiprocessor system design, major unique aspects of our approach in this chapter are the capability to decompose CAL actors based on their formal action- and port-based semantics, and to construct and subsequently transform SSRs and SSR actors from these decomposed representations. As

a result, our methodology has access to and is capable of exploiting the detailed formal modeling semantics of the CAL language, which includes formal modeling of both communication between actors, as well as computations and state transitions within actors. Additionally, our methods provide a novel framework of quasi-static scheduling in terms of SSR actors.

3.3 Analysis framework

Our method to optimize implementation of DSP applications combines the advantages of three complementary tools, as shown in Figure 4.5. The given DSP application is initially described as a CAL network that is composed of CAL actors. The CAL-based dataflow representation is then translated into a DIF-based intermediate representation for analysis by TDP. This TDP-driven analysis produces a set of SSRs, and an associated quasi-static schedule, which is then translated into a reformulated CAL specification. This transformed CAL code is then translated to a C code implementation using CAL2C. The generated CAL2C implementation is optimized to exploit the static structure provided by the SSRs and their enclosing quasi-static schedule.

A CAL actor can in general have two kinds of interfaces — *input ports* and *output ports*. A CAL actor performs computations in sequences of steps, where each step is called an *action*. There are one or more actions associated with a given actor, and an invocation of an actor corresponds to exactly one action. In each action, the actor may consume tokens from its input ports, and may produce tokens on its output ports. Also, there can be one or more *state variables* associated with an actor, and these state variables

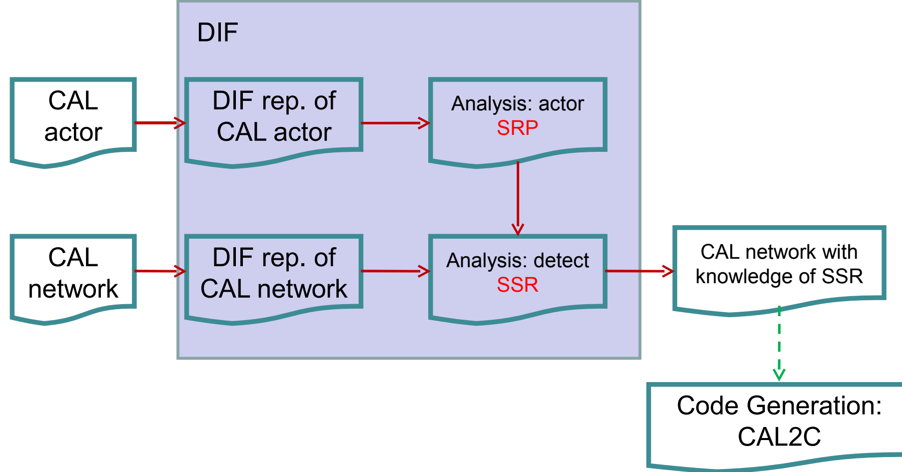


Figure 3.2: Outline of method for optimizing dataflow graph implementation.

can be modified by any action.

We introduce some notation to allow for more detailed discussion of CAL semantics. For simplicity, we assume here that there is exactly one state variable associated with a given CAL actor, but this is not a general restriction of the CAL language — CAL actors can have no state variables or multiple state variables.

A CAL actor A can be represented as a 4-tuple $< \sigma_0, \Sigma(A), \Gamma(A), \succ >$, where $\Sigma(A)$ is the set of all possible values for the state variable; $\sigma_0 \in \Sigma(A)$ is the initial state; $\Gamma(A)$ is the set of all possible actions for actor A; and $\succ$ is a non-reflexive, anti-symmetric and transitive partial order relation on $\Gamma(A)$ called the *priority relation* of A. Intuitively, if $l, m \in \Gamma(A)$, then $l \succ m$ means that l has priority over m if both are "competing" for the next invocation A.

We refer to the set of ports in A as the port set of A, denoted as $ports(A)$. For a given action $l \in \Gamma(A)$, the set of ports that can be affected by the action is denoted (allowing a minor abuse of notation) by $ports(A)_l$. In CAL, different actors can have identically-named ports. To distinguish between identically-named ports in different ac-

tors, we prefix the name of the port with the containing actor, as in $A.a$ and $B.a$. Given a CAL actor A, $inputs(A)$ denotes the set of input ports of A, and $outputs(A)$ denotes the set of output ports of A. Furthermore, given an action $l \in \Gamma(A)$, we again employ a minor abuse of notation, and define $inputs(A)_l = inputs(A) \cap ports(A)_l$, and $outputs(A)_l = outputs(A) \cap ports(A)_l$. These represent, respectively, the sets of actor input and output ports that appear in the action l.

A *guard* is a condition that must be satisfied before the next action in a CAL actor can proceed to execute. In general, a guard condition can involve the actor inputs and actor state. If execution of an action has an associated guard condition, we say that the action is *guarded*. Intuitively, an action that is not guarded executes unconditionally as soon as it is the next action visited during the execution of the enclosing actor A. Also, we say that an action is a *state-modifying* action if the action may, depending on the current state and actor inputs, change the value of the actor state. Given a guarded action m of an actor A, we say that m is *state-guarded* if the guard condition associated with m depends on the value of the state variable associated with A.

Describing an actor in CAL involves describing not only its ports, but also the structure of its internal state, the actions it can perform, what these actions do (such as token production and token consumption, and updating of actor state), and how to determine the action that the actor will perform next.

3.4 Derivation of Statically Schedulable Regions

Our approach for deriving statically schedulable regions involves partitioning and grouping actor ports based on relationships that pertain to various kinds of interactions between ports.

This overall process of partitioning and grouping begins at the level of individual actors. Ports inside an actor can be viewed as having different kinds of associations with one another. Some ports can be viewed as related because they are involved in the same action, while some are related because they affect the same state variable. In this chapter, we apply the following two kinds of port associations:

1. $\exists(l \in \Gamma(A))$ such that $a, b \in ports(A)_l$;
2. $\exists l, m \in \Gamma(A)$ such that $a \in ports(A)_l$, $b \in ports(A)_m$, l is a state-changing action, and m is a state-guarded action.

We define these two conditions as the *coupling relationships*, and we observe that in general, two distinct ports can satisfy zero, one or both of the coupling relationships. Intuitively, if neither of these two conditions is satisfied by two given ports, we separate the two ports into different partitions. If one or both of these conditions is satisfied by two ports of the same actor, then we include the ports in the same partition.

Given two distinct ports a and b of a CAL actor A, we say that a and b are *coupled ports* if they satisfy exactly one or both of the coupling relationships.

Partitioning across ports from different actors is based on connections in the enclosing CAL network. If ports of distinct actors are connected in the CAL network, then

they are combined into the same partition, including any other subsets of ports within the same actors that satisfy coupling relationships with respect to the ports.

After partitioning is performed on actor ports, we perform the grouping phase of our transformation methodology. The sets of ports obtained from partitions are grouped together in an attempt to build larger subsets of computations that can be scheduled statically with respect to one another. In general, static scheduling methods can be used to schedule the computations within such groups, while coordination of each group with the rest of the CAL network can be scheduled dynamically.

There are three kinds of intermediate graphs that are constructed and analyzed during the process of SSR derivation. Two of these are constructed separately for individual actors, and the third intermediate graph is a representation on the overall CAL network.

Partitioning begins from individual actors. The CAL actor is originally represented as a CAL file. The necessary information is translated into a TDL file. From the resulting TDL file, we construct the *coupling relationship graph (CRG)* of an actor A by instantiating a vertex v_p for each port p of A, and an edge (v_a, v_b) for each pair of coupled ports a and b.

Figure 3.3 shows an illustration of coupled ports and CRGs. Here the CRGs for two actors A and B are superimposed in the same graph along with edges between communicating ports of A and B. From the illustration, we see, for example, that the following port-pairs are coupled: $\{A.a, A.x\}$, $\{A.b, A.y\}$, $\{B.a, B.x\}$, $\{B.b, B.x\}$, and $\{B.c, B.y\}$.

The *weakly connected components* of the CRG for an actor A are called *coupled groups*. Weakly connected components are a form of graph structure that can be derived efficiently using well-known graph analysis techniques (e.g., see [55]). Intuitively, in an

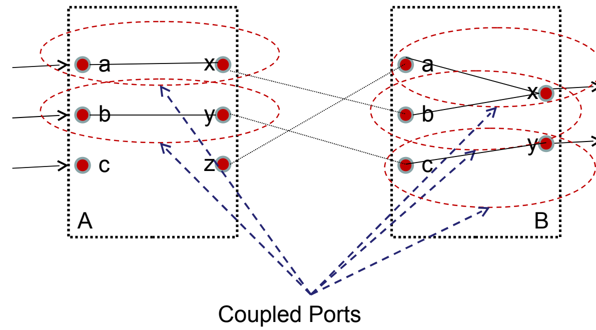


Figure 3.3: An illustration of coupled ports and CRGs.

undirected graph, two actors are in the same weakly connected component if there is a path connecting the two actors. In a directed graph G, two actors are in the same weakly connected component if there is a path that connects the actors in the undirected version of G (i.e., the undirected graph that is derived from G by replacing each directed edge in G with an undirected edge that connects the same pair of actors).

Figure 3.4 shows an example of a directed graph, the undirected version of that graph, the associated weakly connected components.

Figure 3.5 shows an illustration of coupled groups using a similar kind of overall diagram (but based on different actors A and B) as that shown in Figure 3.3.

Once we have partitioned the ports of each actor A into its set C of coupled groups, we examine each coupled group $c \in C$, and we try to extract from c a more specialized kind of port-subset called a *statically-related group* (SRG). In particular, a set of ports

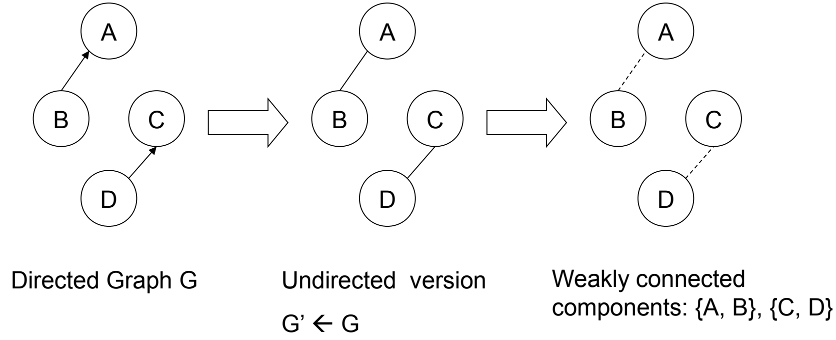


Figure 3.4: An illustration of weakly connected components.

$Z = \{p_1, p_2, \ldots, p_n\}$ within a given coupled group of A is a statically-related group if it satisfies the following three conditions.

1. $\forall l \in \Gamma(A)$, either $Z \subseteq ports(A)_l$, or $Z \bigcap ports(A)_l = \emptyset$, where $\emptyset$ denotes the empty set.

2. Each input port $p_i \in Z$ is a *static rate input port* — that is, there exists a fixed positive integer $cns(p_i,)$ that characterizes the number of tokens consumed from p_i. In other words, for any l such that $p_i \in ports(A)_l$, we have that exactly $cns(p_i,)$ tokens are consumed from p_i during l.

3. Similarly, each output port $p_j \in Z$ is a *static rate output port*, which means that there exists a fixed positive integer $prd(p_j,)$ that characterizes the number of tokens produced onto p_j, regardless of which "containing action" is being executed.

We say that a port is a *static rate port* if it is either a static rate input port or a static rate output port.

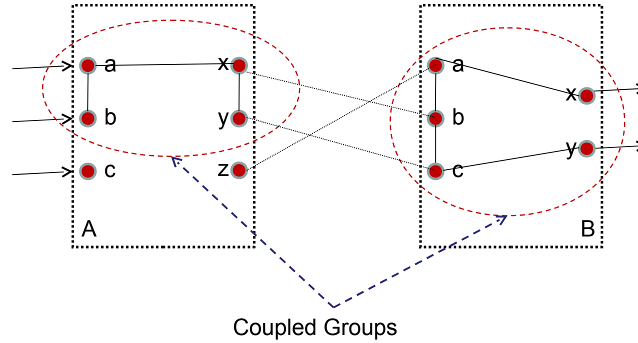


Figure 3.5: An illustration of coupled groups.

SRGs (statically-related groups) can be derived by constructing and analyzing an intermediate graph representation that we call the *static relationship graph*. Given a coupled group $R = a_1, a_2, \ldots, a_n$, we construct the static relationship graph of R by first instantiating a vertex x_{a_i} for each $a_i \in R$ such that a_i is a static rate port, and a vertex v_z for every action z in the actor. We then instantiate an edge (x_{a_i}, v_z) for every ordered pair (a_i, z) such that $a_i \in ports(z)$. By definition, the static relationship graph is a bipartite graph. Figure 3.6 shows an example of a static relationship graph and the statically-related group derived from Figure 3.5.

The SRGs of an actor can be derived by computing the weakly connected components of the static relationship graph — each weakly connected component of the static relationship graph is an SRG.

Once the SRGs have been determined, we construct another intermediate graphical

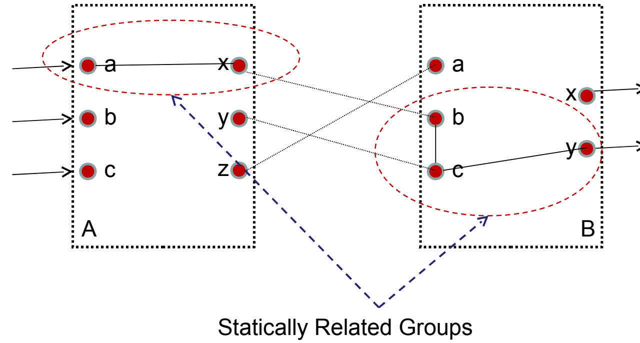


Figure 3.6: An illustration of a statically-related group.

representation, which we call the *SRG graph*. SSR detection then operates directly on the SRG graph.

Before defining the SRG graph, however, it is useful to define the concept of connectivity between SRGs. Given two SRGs A_1 and A_2, we say that A_1 and A_2 are *connected* if there exist ports p_1 and p_2 such that $p_1 \in A_1$, $p_2 \in A_2$, and p_1 and p_2 are connected by an edge in the enclosing CAL network (i.e., p_1 and p_2 are communicating ports in the overall CAL specification).

The process of SRG graph construction can now be described as follows. We construct the SRG graph of a given CAL network by instantiating a vertex v_S for each SRG S in the graph, and instantiating and edge v_S, v_T for every pair S, T of SRGs that are connected.

Once the SRG graph has been constructed, the SSRs (statically schedulable regions)

can be derived through another computation of weakly connected components. In particular, suppose that $X_1, X_2, \ldots, X_n$ are the weakly connected components of the SRG graph. Thus, from the definitions of the SRG graph and weakly connected components, each X_i can be expressed as a set

$$X_i = \{x_{i,1}, x_{i,2}, \ldots, x_{i,m_i}\}, \tag{3.1}$$

where each $x_{i,j}$ represents the jth SRG within the ith weakly connected component of the SRG graph.

The SSRs of the given CAL network can then be expressed formally as the set $R = \{r_1, r_2, \ldots, r_n\}$, where for each i, r_i is defined by

$$r_i = \bigcup_{j=1}^{m_i} x_{i,j}. \tag{3.2}$$

Each $r \in R$ is called a statically schedulable region (SSR) of the given CAL network.

Figure 3.7 shows an example of an SRG graph and the obtained statically schedulable region.

3.5 Scheduling of SSRs

After deriving the SSRs from a given CAL network, a natural next step is scheduling the SSRs — i.e., determining the execution order of the computations in each SSR. Since, by construction, each SSR is statically schedulable, we can efficiently adapt SDF scheduling techniques for this step in our proposed design flow.

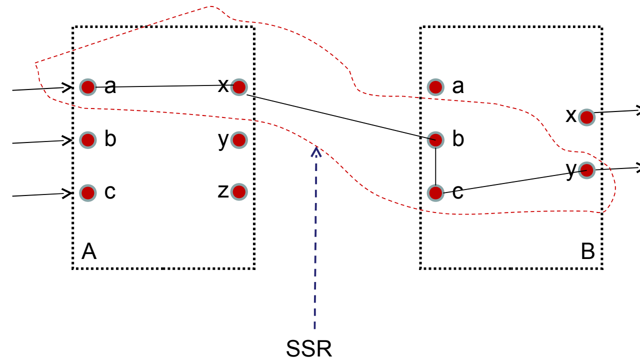


Figure 3.7: An illustration of a statically schedulable region.

In order to apply SDF scheduling techniques to an SSR, we first construct a set of one or more SDF actor actors from the ports in the SSR. In particular, all of the ports of a given actor A within an SSR s are combined to form a corresponding *SSR actor* $\sigma(s, A)$. Note that in general, $\sigma(s, A)$ may contain all of the ports in A or a proper subset of the ports, depending on whether all of the ports of A are in s.

After decomposition of an SSR into SSR actors, an SDF graph representation of the SSR emerges naturally, and SDF scheduling techniques can be applied to this SDF graph representation to derive a static schedule for the SSR.

Note that in general, an SSR actor can correspond to the full functionality of a single actor in the overall CAL network, or it can correspond to only part of the functionality. Typically, the latter applies. Furthermore, the same CAL actor can have associated SSR actors in different SSRs.

3.5.1 IDCT Example

Figure 4.7 illustrates SSRs within an IDCT (inverse discrete cosine transform) subsystem. Here, the main body of the IDCT is composed of the actors *row*, *tran*, *col*, *retran*, and *clip*. The *dataGen* and *print* actors are used to provide a testbench for the network — *dataGen* is responsible for generating input data, and *print* for displaying the output from the IDCT computation. The shaded regions shown in the figure correspond to the different SSRs, which are unique to the application.

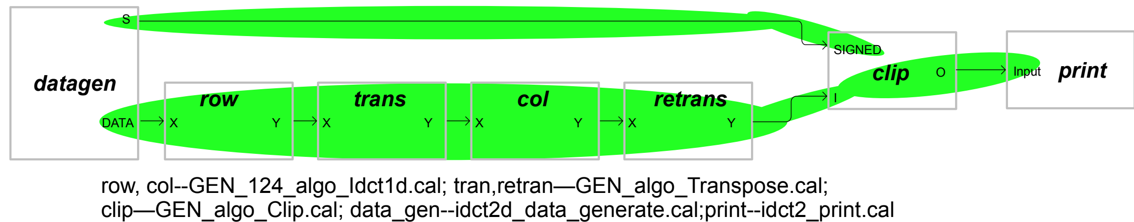


Figure 3.8: SSRs in the IDCT subsystem.

Each SSR can be scheduled quasi-statically, which means a significant portion of the schedule structure can be fixed at compile time. When we map the enclosing application onto a multi-core platform, each SSR can be allocated to a single core, and the scheduling for each SSR can be controlled on the core that is allocated to the SSR. If the granularity of some SSRs is so large that allocating them as single-processor subsystems results in poor load balancing, the SSR detection process can be post-processed with a load-balancing phase that optionally adjusts SSR granularity to improve overall schedule performance. Such refinement of SSRs before allocation is a useful direction for further investigation.

If we map the IDCT onto a dual-core system based on SSR analysis, a straightfor-

ward mapping for this case is shown in Figure 4.7. In this case, the connections between the cores are connections inside both the *dataGen* and *clip* actors. These weak connections can be implemented using semaphore primitives. Furthermore, inside each core, the actions can be statically scheduled in terms of checks on an appropriately defined semaphore. Here, we can easily take advantage of well known SDF scheduling techniques, such as APGAN [56, 57], which provides a framework for incremental schedule construction that can be adapted to a variety of objectives.

An example of SSR scheduling for the IDCT example is shown in Figure 4.9. Here the schedule for a single SSR is represented in the form of a *schedule tree*. This schedule tree representation corresponds to a nested loop schedule where the internal nodes of the tree correspond to loops; the iteration counts of these loops are given by the labels of the corresponding internal nodes; and leaf nodes of the tree correspond to SSR actors. More details on and applications of this kind of schedule tree representation can be found in [58].

In the schedule tree shown in Figure 4.9, SSR actors that are labeled with purely alphabetic names (no number in the name), such as *tran* and *row*, indicate SSR actors that correspond to the entire computation of the associated CAL actor. On the other hand, SSR actors whose names contain numbers correspond to actors in the CAL network that map to multiple SSR actors across multiple SSRs.

Note also that for this IDCT example, every actor port is contained in an SSR actor. In general, some ports may lie outside of all SSRs; we refer to such ports as *dynamic ports*. However, for the IDCT example, there are no dynamic ports.

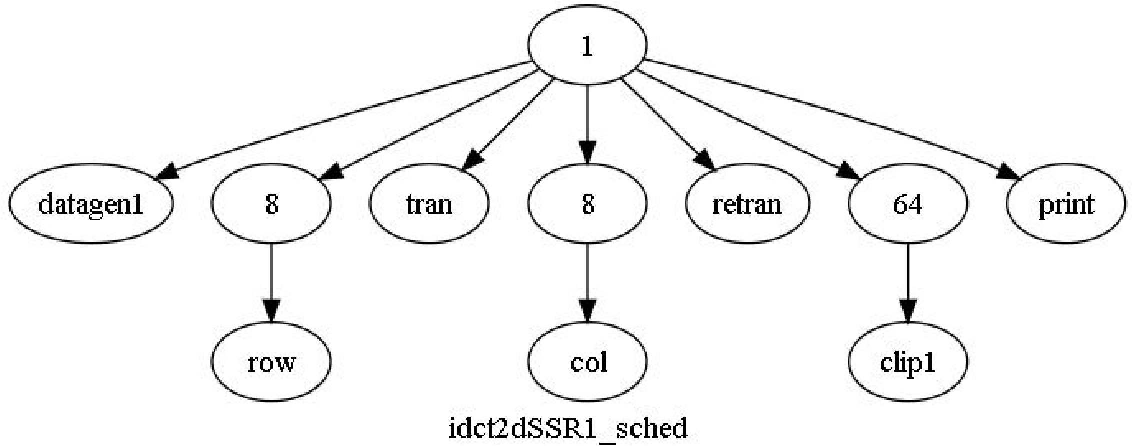


Figure 3.9: Schedule tree for an SSR in the IDCT example.

3.5.2 Simulation Results

After integrating results of SSR analysis into CAL2C, we obtained a modified version of CAL2C, which we call *CAL2C-SSR*. To evaluate the effectiveness of our SSR techniques, we conducted experiments on a dual-core 2.5Ghz computer. We generated C code using CAL2C and CAL2C-SSR for three different *IDCT* versions. The first version (V1) does not employ any SSR analysis, and can be viewed as being scheduled purely through SystemC, which is used as the default scheduling mechanism in CAL2C.

The second version (V2) uses CAL2C-SSR. This version exploits the SSRs illustrated in Figure 4.7, and employs a quasi-static integration of static schedules for these SSRs with top-level dynamic scheduling. In this version, two SSRs are mapped onto two cores, and semaphore primitives are used for inter-SSR communication.

The third version (V3) also uses CAL2C-SSR. This version also uses a modified, more predictable version of the *clip* actor that can be used when the input data is known in advance. In the new version of *clip*, the ports *Signed* and *O* are rewritten to become

coupled ports. Then the original two SSRs are combined as one SSR through connections inside *clip*. In the illustration of V3 shown in Figure 4.10, the *IDCT* system becomes an SDF model that runs as a single thread. Since entirely static scheduling is used in this version, V3 is the most efficient in terms of execution speed.

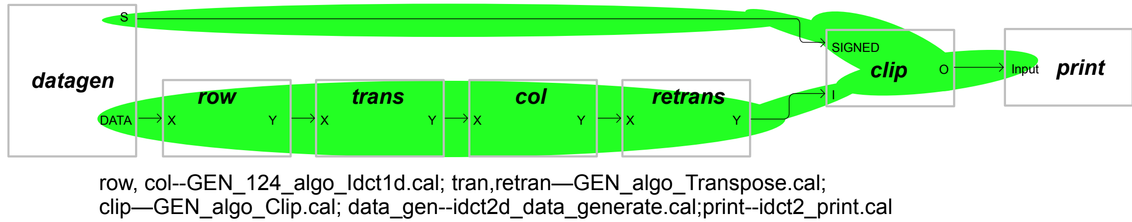


Figure 3.10: IDCT subsystem with a single SSR.

We experimented with all three *IDCT* versions using Microsoft Visual Studio. The results are shown in Figure 4.11. Here, V2 shows an improvement in performance of 1.5 times compared to V1, whereas V3 shows the best performance among all three versions.

Note that while V3 exhibits the best performance, demonstrates that larger SSR regions can lead to significant improvements in performance, and is generally interesting as a kind of “limit study,” this version is not of practical utility. This is because V3 requires prior knowledge of input data, which is not a practical assumption for real-time operation.

3.6 Grouping of Dynamic Ports and SSRs

In this section, we explore a new form of dataflow graph analysis to help streamline the interaction between dynamic ports and SSRs. Such analysis helps to improve the efficiency of SSR-based quasi-static schedules.

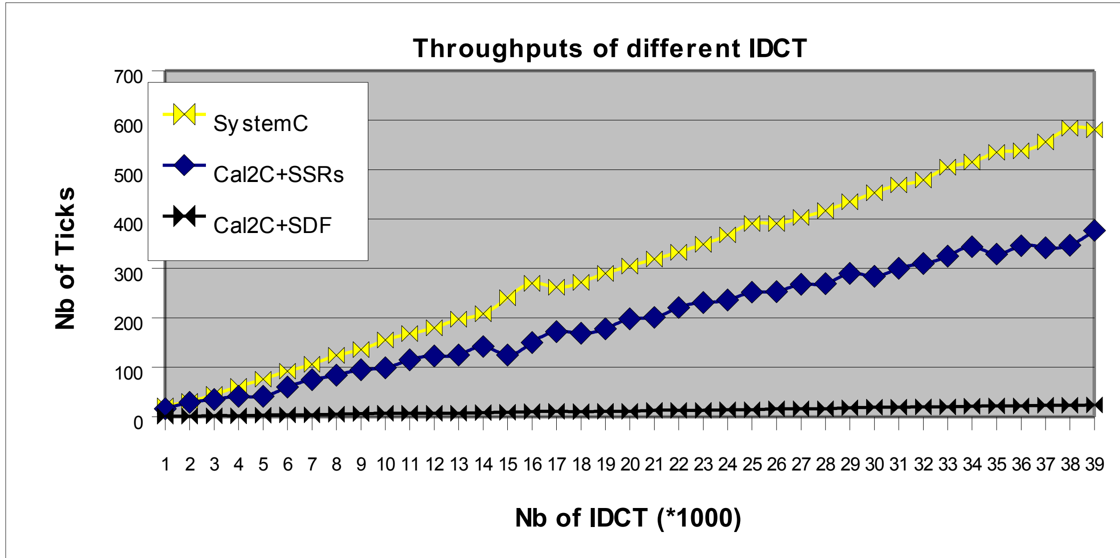


Figure 3.11: Results: clock cycles vs number of iterations.

Recall that a port of a CAL network that is not contained in an SSR is called a *dynamic port*. Given a dynamic port p, an SSR s, and an action a in s (i.e., a is part of one of the SSR actors within s), we say that p *is related to* s if (1) p is referenced in the body of a; (2) p is referenced in the action guard of a; or (3) p outputs tokens to a (i.e., there is an input port that consumes tokens produced from p whenever a fires). We define the *strength of the relationship* between the dynamic port p and the SSR s, denoted $\Sigma(p, s)$, as the total number of actions in s that p is related to. Thus, in general, $\Sigma(p, s)$ is a non-negative integer that is bounded above by the total number of actions in s.

In this section, we explore a scheme by which dynamic ports are grouped together with SSRs based on the "strength" metric Σ. We refer to this scheme as *strength-based, iterative grouping (SBIG) of dynamic ports and SSRs*. To demonstrate this approach, we select a port-SSR pair $\Sigma(p_1, s_1)$ that maximizes the strength value $\Sigma(p, s)$ over the set of all port-SSR pairs. Then we remove p_1 from further consideration, and select a port-SSR

pair $\Sigma(p_2, s_2)$ that maximizes the strength value over all remaining dynamic ports and all SSRs. Then we remove p_2 from further consideration, and continue this process of matching up SSRs successively with dynamic ports until every dynamic port has been assigned to an SSR. This leads to a partitioning of the set of dynamic ports across the set of SSRs.

At this point, each dynamic port is grouped with exactly one SSR, and in general, each SSR is grouped with zero or more dynamic ports. The dynamic ports are then analyzed to conditionally schedule the SSRs that are grouped with them. The results of these conditional schedule constructions are then combined to form the quasi-static schedule for the overall CAL network.

We experimented with our strength-based, iterative grouping approach on the MPEG-4 RVC SP decoder system shown in Figure 4.2. When applied to this system, our tools for SSR detection derived a total of 30 SSRs. 32 ports are left outside the SSRs — these are the dynamic ports. By applying our method of strength-based, iterative grouping, we partitioned the 32 dynamic ports across the set of available SSRs. We then used the resulting partitioning result to derive a quasi-static schedule for the system.

For these experiments, we further modified the scheduler in CAL2C [9] to better accommodate SSRs. All of the SystemC primitives have been removed from the current version of Cal2C. The current scheduler is a round robin scheduler executing each actor in a loop; an actor is fired until input tokens are available and output FIFOs are not full. SSRs can easily be incorporated in this fully software-based implementation, independent from SystemC, by removing all of the tests on the FIFOs.

A code generator that translates CAL-based dataflow models to SystemC is pre-

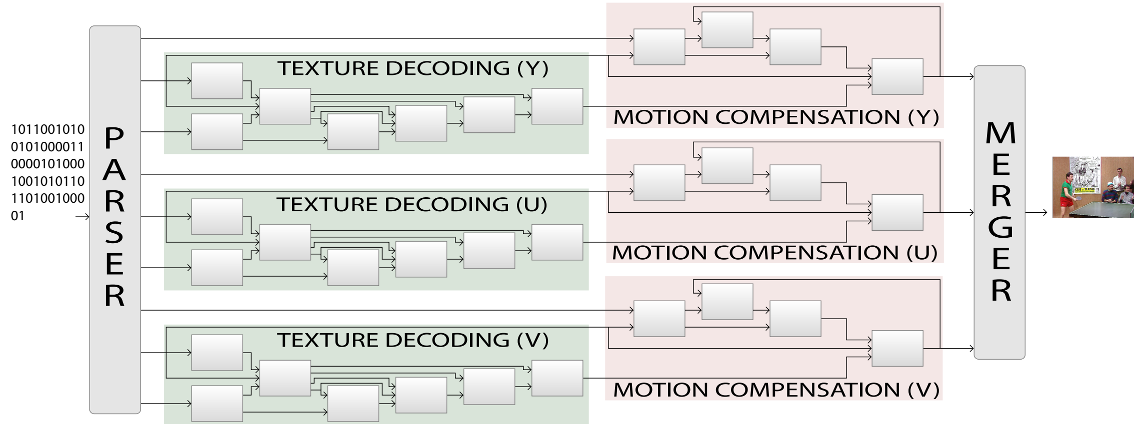


Figure 3.12: A block diagram of an MPEG RVC decoder.

sented in [9]. Such a tool can be useful for simulation, but may lead to major inefficiencies if targeted to actual implementations. For example, in such a translation approach, each actor in SystemC is executed in its own thread. Thus, context switches can occur frequently during execution, and this can lead to poor performance, especially if many actors with low granularity are present.

Compared to a direct translation in SystemC [9], our C mono-thread implementation is indeed 4 times faster. For our multi-core implementation, we have statically mapped the actors (each actor is assigned a priori to a core). For each core, actors assigned on it are turned into a single thread with its own dataflow process network scheduler. Since only one thread is executed on each core, threads are not executed concurrently but in parallel.

We conducted experiments involving the applications of both CIF sequences with size 352x288 and sequences with size 624x352. A CIF-size image (352x288) corresponds to 22x18 macroblocks. As shown in Table 3.1 and Table 3.2, the experimental results demonstrate that CAL2C with quasi-static scheduling using strength-based, iterative

grouping (SBIG) on the round robin scheduler has the best performance in a multi-core system. CAL2C with SBIG can be applied to more applications besides MPEG, and this is a useful direction for future work.

SP decoder	speed(frame/second)
monoprocessor with SystemC scheduler	8
monoprocessor with round robin scheduler	42
monoprocessor with round robin scheduler and SSRs	44
dualcore processor with round robin scheduler and SBIG	50

Table 3.1: MPEG-4 SP decoder performance of CIF sequence 352x288.

SP decoder(with round robin scheduler)	speed(frame/second)
monoprocessor	10
monoprocessor with SSRs	11
dualcore processor	15
dualcore processor with SBIG	16

Table 3.2: MPEG-4 SP decoder performance of sequence 624x352.

We note that the process of strength-based, iterative grouping (SBIG) between dynamic ports and SSRs, as well as the derivation of SSRs, are fully automated processes in our experimental setup. However, the output of SBIG is presently converted manually into a corresponding quasi-static schedule for the given CAL network. Automating the connection between SBIG and quasi-static scheduling, as well as exploring new techniques to further optimize the resulting schedules are useful directions for further study.

Chapter 4

Exploring the Concurrency of an MPEG RVC Decoder

4.1 Overview

In this chapter, we apply our proposed framework to video processing systems. We present how dataflow techniques optimize the existing system by exploring the concurrency of MPEG video processing systems, as shown as red, bold and italic in Figure 4.1. Our work presented in this chapter is also available in the publication [22].

Upcoming MPEG video coding standards are intended to increase the quality and the flexibility of complex and versatile future video coding applications. Since 1988, several MPEG standards have been developed successfully based on available hardware technologies and software support. Early MPEG standards (MPEG-1 and MPEG-2) were specified by textual natural-language descriptions. Starting with MPEG-4, reference software written in C/C++ became the formal specification of the standard. Written in a sequential programming language, this reference software describes a sequential algorithm,

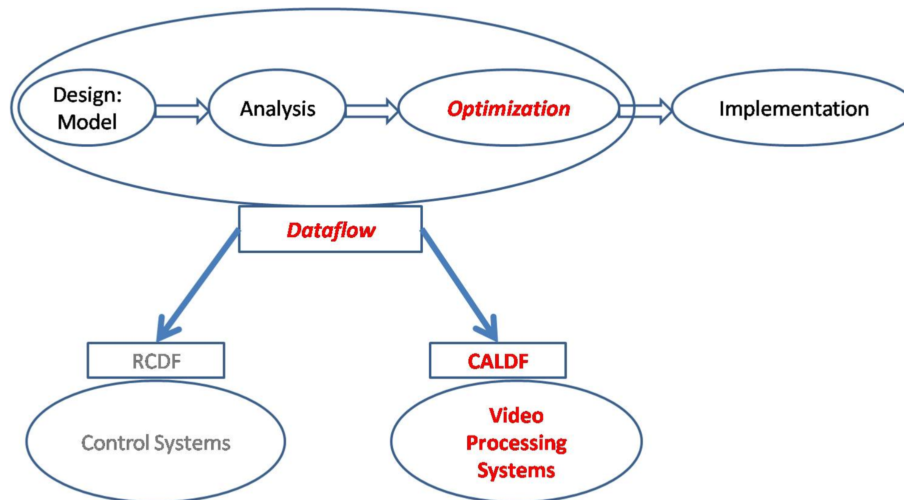


Figure 4.1: Framework for video processing systems: Optimization.

effectively hiding the considerable inherent concurrency of a video decoder. Furthermore, the reliance on global memory and state makes the reference description difficult to modularize, resulting in a very monolithic specification. The observation of these drawbacks of current video standard specification formalism led to the development of the Reconfigurable Video Coding (RVC) standard [48]. The key concept of RVC is to be able to design a decoder at a higher level of abstraction than the one provided by current generic monolithic C based specifications to express the potential parallelism of the decoder. Furthermore, hardware for embedded systems employs increasing amounts of parallelism — e.g., in platforms such as multi-core systems on chip. When starting from sequential specifications (e.g., in C/C++), designers targeting parallel platforms typically have to start with a complete rewrite of the reference code. This scenario leads to the following questions: What are suitable languages for developing implementations on parallel platforms? How is application concurrency represented and exploited? How can designers enhance application concurrency?

CAL, as a dataflow/actor-oriented language, is a promising answer to the first question and has been chosen by MPEG RVC as the normative language to describe MPEG decoder coding tools. In addition to a stronger encapsulation of coding tools and a more explicit description of the parallelism inherent in a decoding algorithm, constructing decoding algorithms as dataflow networks creates the opportunity to apply the wide range of techniques for analyzing and implementing dataflow systems that have been developed in the past (e.g., see [8]). Furthermore, CAL has been designed to make explicit a number of relevant properties of dataflow actors, which can be extracted and used as input to those techniques. Concurrency mainly benefits system execution speed, especially for real time systems such as video decoders. There are other issues, such as memory/buffer and energy efficiency, related to concurrency, which are beyond the scope of this chapter, and are useful directions for future work.

References [48], [46], [59] cover related aspects of reconfigurable video coding and CAL-oriented tools. In particular, [48] gives an overview of the overall RVC framework; Reference [46] provides details on the software code generator CAL2C; and Reference [59] elaborates on a hardware code generator for CAL. In contrast, this chapter is distinctive in it focus on analyzing concurrency and exploiting parallelism; the topic of concurrency is not addressed in depth in References [48], [46] and [59].

Using CAL as a concrete design representation framework, this chapter places emphasis on answering the last two questions described above. More specifically, this chapter analyzes data parallelism and pipeline concurrency that are exposed by CAL actors. Furthermore, we exploit these forms of concurrency with new techniques for cross-actor optimization. These techniques are enabled by dataflow analysis on intermediate repre-

sentations that are derived from CAL specifications. Based on these ideas, we present novel tools and techniques for efficient implementation of video processing systems on multi-core platforms.

Section 4.2 introduces previous work related to advanced reconfigurable video coding technology, dataflow models, and the CAL language. multi-core systems are also discussed in this section. Section 4.3 analyzes inter-actor concurrency obtained from CAL specifications from the viewpoint of both hardware and software implementation. Section 4.4 proposes techniques for cross actor-optimization that enhance multi-core system performance. Simulation results are also presented in this section.

4.2 Background

4.2.1 Reconfigurable video coding

The desire for a more compositional approach for building existing and future video standards, and for a shorter path to parallel implementation has led to the development of the reconfigurable video coding (RVC) standard [48]. The MPEG RVC framework is a new standard under development by MPEG that aims at providing a unified high-level specification of current and future MPEG video coding standards. Rather than building a monolithic piece of reference software, RVC standardizes an “Abstract Decoder Model” (ADM) composed of a network that interconnects a set of video coding tools with uniform interfaces extracted from a library. Decoder descriptions are composed from that library, which permits a wide range of decoding algorithms.

The MPEG RVC framework is currently under development in MPEG as part of the MPEG-B part 4 [60] and MPEG-C part 4 [61] standards. The abstract decoder is built as a block diagram or network in which blocks define processing entities called functional units (FUs) and connections represent the data path between the FUs. This network is described in MPEG-B part 4 as an XML dialect called FU Network Language (FNL). RVC also provides in MPEG-C part 4 a normative standard library of FUs, called the “Video Tool Library (VTL)”, and a set of decoder descriptions expressed as networks of FUs. CAL is currently chosen as the language to express the behavior for the coding tools of the library (VTL). Such a representation is modular and helps in formulating the potential configuration of decoders in terms of modifications of network topologies. The ADM is a CAL dataflow program that constitutes the conformance point between the normative RVC specification and all possible proprietary implementations that have to be generated to decode the incoming bitstreams. Thus the MPEG RVC standard leaves open the platforms and the implementation methodologies that can be used to generate any RVC proprietary implementation. This provides all possibility of generating parallel and concurrent implementations for a wide variety of existing and emerging implementation platforms. Thus, indirect generations of implementations will be possible together with the direct synthesis of software and hardware from the ADM. All these possibilities enable, for each application scenario, the users to select the most appropriate implementation methodology.

4.2.2 Concurrency

In computer science, concurrency is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other. The computations may be executing on multiple cores in the same die, preemptively time-shared threads on the same processor, or executed on physically separated processors.

As mentioned before, real-world embedded applications are typically developed in sequential programming languages, such as C/C++. In addition to CAL, various other languages have been developed for concurrent programming. An example of such a language is the Erlang language [62]. Many of the previously-developed concurrent programming languages, including the Erlang language, are oriented towards general-purpose programming. In contrast, CAL targets more specialized application domains, such as video processing and many other domains of DSP, that are suited to dataflow representations.

4.2.3 Multi-core systems

Multi-core devices, which incorporate two or more processors on the same integrated circuits, are becoming increasingly relevant to the design and implementation of DSP systems (e.g., see [63]). In multi-core platforms, all cores can execute instructions independently and simultaneously. While instruction level concurrency is targeted by single core processors, multi-core structures target task level concurrency.

In multi-core platforms, carefully managing communication and synchronization among different cores is important to achieve efficient implementations. Two or more processing cores sharing the same system bus and memory bandwidth limit the achiev-

able performance improvements. For example, if a single core is close to being memory-bandwidth-limited, going to a dual-core solution may only result in 30% to 70% improvement. If memory bandwidth is not a problem, 90% or greater improvement can be achievable. It is possible for an application that used two CPUs to end up running significantly faster on a single dual-core platform if communication between the CPUs was the limiting factor.

The ability of multi-core processors to increase application performance depends on the use of multiple concurrent tasks within applications. Therefore, if code is written in a form that facilitates decomposition into concurrent tasks, the multi-core technologies can be exploited more effectively. In the context of dataflow programming, the CAL language is suitable for such decomposition into concurrent tasks. This chapter addresses the systematic mapping onto parallel platforms of concurrent tasks that are extracted from CAL programs.

4.3 Inter-actor concurrency analysis

4.3.1 Data-driven processing

The transitions between actions within an actor are purely sequential: actions are fired one after another. This means that during each actor invocation, only one action is executed inside the actor. In a CAL network, distinct actors are functionally independent and work concurrently, with each one executing its own sequential operations based on the availability of sufficient numbers of tokens on actor input ports.

Connections between actors in CAL are purely data-driven. This data-driven property of CAL results from two properties: A CAL actor executes only if there are enough tokens on the actor input ports to trigger an action, and execution of a CAL actor produces nothing "outside the actor" other than tokens on the output ports of the actor. In other words, CAL actors communicate with one another only using tokens that are passed along dataflow graph edges. Networks of CAL actors are described in FNL language.

The CAL language naturally supports hierarchical design, which is important for MPEG RVC coding systems. In hierarchical dataflow graphs, actors can have their internal functionality specified in terms of embedded (nested) dataflow graphs. Such actors or FUs are called *hierarchical actors* or *super actors*. A hierarchical actor in CAL can be specified in terms of a network of CAL actors. This approach facilitates modularity, where the internal specification of any actor can be modified without impacting that of other actors.

In this chapter we target as a case study the example of an MPEG-4 simple profile decoder (*MPEG-4 SP decoder*) described in RVC formalism. A graphical representation of the macroblock-based SP decoder description is shown in Figure 4.2. In Figure 4.2, the shaded area indicated as *texture decoding* represents a super actor that is described in FNL . Similarly, the shaded area labeled as *motion compensation* also represents a hierarchical actor in our design. Furthermore, inside the actor *texture decoding*, the *Inverse DCT* actor represents a lower-level super actor, which is also described in FNL and is composed of several *atomic* (non-hierarchical) actors/FUs. The other blocks in the diagram are atomic actors/FUs.

Overall, in the MPEG-4 SP decoder shown in Figure 4.2, there are three hierarchies

and atomic actors and super actors from different hierarchies are interleaved. Note that for readability, only one edge is shown in cases where two actors are connected by more than one edge. It is possible, for example, that multiple edges connect the same pair of actors because of connections between different interfaces of hierarchical subsystems.

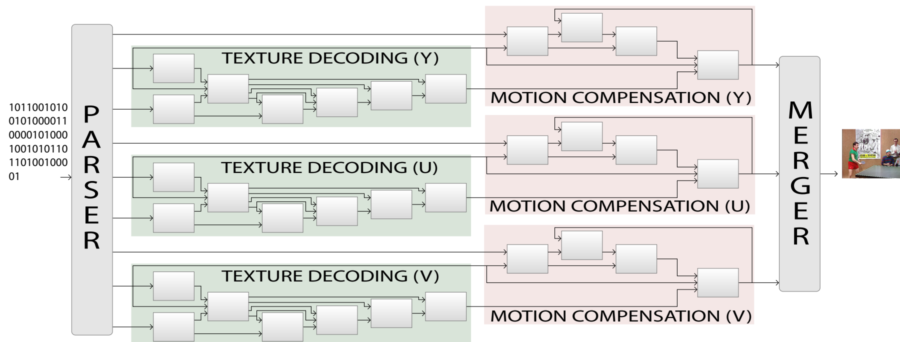


Figure 4.2: An RVC block diagram of an MPEG-4 Simple Profile decoder.

4.3.2 Data parallelism inside CAL networks

In Figure 4.2, there are three sub-systems that handle Y, U and V separately. These three sub-systems share the same set of processing modules in the form of CAL actors that differ only in their associated sample rates.

The structure of a macroblock demands that the processing used in MPEG-4 utilize 4:2:0 YUV processing. The color channels sample at exactly half the rate in both the horizontal and vertical directions as they relate to the luminance (Y) channel. For this reason, for every U and V pixel, there are four Y pixels. The spatial relationship among the three channels is documented in many MPEG articles.

The subsystems for Y, U and V are concurrent in the sense that they handle signals from different channels. These signals are generated by the *parser* actor, and then are

directed to the Y, U, and V subsystems for processing. In this way, the CAL network explicitly exposes inter-actor, and inter-subsystem concurrency in the overall application.

4.3.3 Pipeline concurrency analysis

Exploiting different forms of concurrency is often important when we implement DSP applications on multi-core systems. The intrinsic capability of CAL operators and programming constructs to describe different forms of concurrency, including pipeline concurrency, which is a special form of task level concurrency for consecutive input data, and more irregular forms of task level concurrency, makes CAL especially useful for design and implementation of DSP applications.

Each atomic CAL actor encapsulates a set of computations that are executed sequentially — i.e., there is no concurrency among different actions at the intra-actor level. However, the data-driven semantics of CAL actors, where different actors can execute whenever they have sufficient input data, effectively exposes inter-actor concurrency. How effective a CAL representation is in exposing inter-actor concurrency depends not only on the CAL semantics but also on the particular CAL program that is used. Given a CAL program, it may be possible to redesign the program to expose more concurrency; such rewriting of CAL programs is beyond the scope of this chapter.

Our CAL representation for the MPEG-4 SP decoder is composed of 27 distinct actors. Some of these actors are instantiated multiple times; the total number of actor instantiations in our MPEG-4 SP decoder program is 42. If a multi-core platform with enough processing cores is available, each actor instance can be mapped to a separate

core, and we can use the dataflow semantics of inter-actor communication in CAL to drive the communication and synchronization among the multiple processors. If there are not enough processor cores to accommodate such a one-to-one mapping between actor instances and cores, we need to map groups of multiple actors to the same core. Furthermore, even if enough cores are available, it may be desirable to employ such "grouped mappings" (and leave some processors unused) if the overhead of inter-processor communication dominates parallel processing efficiency for some subsystems (e.g., when the granularity of the actors is relatively small).

Thus, grouping of actors onto multiple processing units is in general an important step in the mapping of dataflow programs onto multi-core platforms (e.g., see [16]). This step is often referred to as "actor assignment" (i.e., the assignment of actors to physical processors). To derive efficient parallel implementations of CAL networks, it is generally important to perform actor assignment carefully.

4.3.4 Concurrency from available code generators

A number of code generators have been developed for translating CAL programs into platform-specific implementations.

For example, a hardware description language (HDL) code generator, CAL2HDL, was developed at Xilinx [59]. In the current version of CAL2HDL, an actor with N actions is translated into $N+1$ "threads", one for each action and another one for the *action scheduler*, which coordinates execution across the different actions. The action scheduler is the mechanism that determines which action to fire next. This determination is made

based on the availability of tokens, the guard expression for each action (if present), the underlying finite state machine schedule, and the action priorities. The resulting hardware circuit can be optimized further in a sequence of steps, including bit-accurate constant propagation, static scheduling of operators, and memory access optimization. Detailed discussion of CAL2HDL is beyond the scope of this chapter; we refer the reader to [59] for further information.

HDL programs generated from CAL2HDL provide suitable targets for dedicated hardware implementation and fully concurrent programs. However, targeting CAL to embedded processors, including embedded multi-core platforms, requires a different approach, including different abstractions and target languages.

CAL2C [3, 64] is a code generator that translates CAL into C code, and provides a suitable path for implementing CAL programs on embedded processors. An important objective in the development of CAL2C is the minimization of context switch overhead.

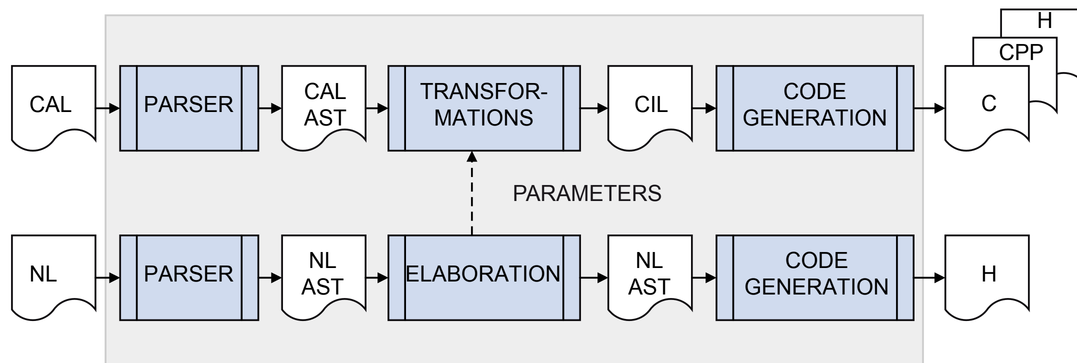


Figure 4.3: CAL2C compilation process: The action translation process starts with an abstract syntax tree (AST) derived from the CAL source code; the transformed CAL AST is expressed in the C intermediate language (CIL) [2], where CAL functional constructs are replaced by imperative ones.

In CAL2C, software synthesis from a CAL network includes two parts: actor transformation [3] and network transformation [64]. Inside an actor, CAL translation is per-

formed in two parts: translation of actor code (actions, functions, and procedures) to express the core functionality, and implementation of the action scheduler (priorities, FSMs, and guards) to control execution of the actions [3]. Translating CAL actor code produces a single C file that contains translated versions of functions, procedures, and actions. Each action is converted into one function and the functions to describe the actions for one CAL actor share a set of common input/output ports as the function arguments in C. An action scheduler is created to control action selection during execution. Priorities, guards, token consumption rates, and FSMs have to be translated to this end. Determining the overall order of action execution is required to have a consistent evaluation of actions that can be fired. SystemC scheduling is used in CAL2C generation as a sequential scheme. Figure 4.3 illustrates how CAL2C works. For further details on CAL2C, we refer the reader to [64].

In [64], we have applied CAL2C successfully on our CAL-based design for the MPEG-4 SP decoder. Simulation results show that the synthesized C-software is as fast as 20 frames/s, which provides near-real-time performance for the QCIF format (25 frames/s) on a standard PC platform. It is interesting to note that our CAL-based speed processing generated from CAL2C is scalable in terms with the number of macro-blocks decoded per second (MB/s) (the number of MB/s remains constant when dealing with larger image sizes). Furthermore, this number can be increased if we use more powerful processors.

Although both forms of design produce code in the same kind of language, code generated from CAL2C is different compared to implementations that use C/C++ as the starting point. As a dataflow language, CAL restricts the way in which designers can

describe applications, and these restrictions carry over through CAL2C to produce code that is more modular and purely dataflow-oriented compared to implementations that are developed directly from C/C++. This is illustrated in Figure 4.4.

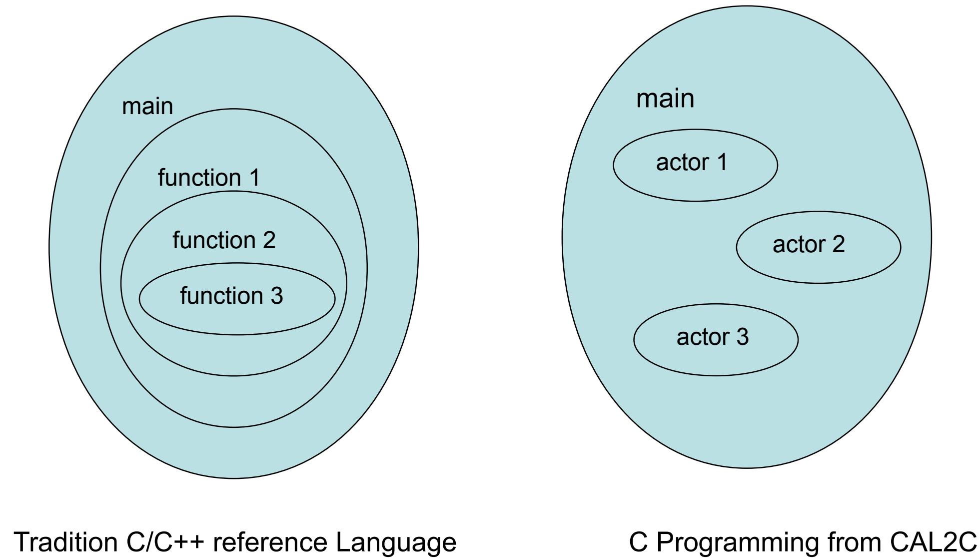


Figure 4.4: Comparison between direct-C/C++-based implementation and implementation using CAL2C.

After obtaining a set of threads from CAL2C, the mapping of these threads onto the targeted multi-core platform remains an important issue. Since CAL-based threads communicate with one another through tokens that pass along dataflow graph edges, one must provide mappings from dataflow edges into appropriate communication primitives, depending on whether the edges (i.e., the incident source and sink actors) are assigned to the same core (intra-core communication) or to different cores (inter-core communication). In general, inter-core communication is less efficient, and this should be taken into account carefully when mapping threads onto cores.

Previous CAL-based synthesis tools, including CAL2HDL and CAL2C, focus on intra-actor code generation without attention to inter-actor optimization. For example,

for CAL2C, both actor- and network-level schedulers are based on run-time scheduling mechanisms from systemC, which is not optimized for cross-actor dataflow scheduling.

In the next section we explore new techniques for inter-actor optimization of CAL programs, and we apply these techniques in conjunction with CAL2C to derive optimized software implementations for multi-core platforms.

4.4 Inter-actor optimization for CAL networks

Although CAL2C exposes task level concurrency, there is significant room for improvement in CAL2C-based implementation in terms of the scheduling mechanisms used to map and coordinate tasks across multiple processors. In particular, since CAL2C inherits the scheduling mechanism of systemC, there is no use of task level static scheduling.

In this section, we describe techniques to exploit the concurrency exposed by CAL network representations. In particular, we develop new graph analysis techniques that result in efficient inter-actor optimization for CAL-based implementations. The result of our optimization is in the form of units of scheduling that we call *statically schedulable regions* (SSRs). SSRs are of significant utility in static scheduling, and mapping of CAL networks onto multi-core systems.

4.4.1 DIF and network analysis capability

In this section, we present our application of the dataflow interchange format (DIF) package [8, 65], a software tool for analyzing DSP-oriented dataflow graphs, to the analysis and transformation of CAL networks for efficient implementations.

The dataflow interchange format (DIF) is proposed as a standard approach for specifying and integrating arbitrary dataflow-oriented semantics for DSP system design. The DIF language (TDL) is an accompanying textual design language for high-level specification of signal-processing-oriented dataflow graphs. The TDL syntax for dataflow graph specification is designed based on dataflow theory and is independent of any design tool. For a DSP application, the dataflow semantic specification is unique in TDL regardless of the design tool used to originally enter the specification. The TDL grammar and the associated parser framework are developed using a Java-based compiler-compiler called SableCC [66]. For the complete DIF language grammar and a detailed syntax description, we refer the reader to [65].

TDL is designed as a standard approach for specifying DSP-oriented dataflow graphs. TDL provides a unique set of semantic features to specify graph topologies, hierarchical design structures, dataflow-related design properties, and actor-specific information. Because dataflow-oriented design tools in the signal processing domain are fundamentally based on actor-oriented design, TDL provides a syntax to specify tool-specific actor information, which ensures that all relevant information can be extracted from a given design tool. The DIF Package (TDP) is a software tool that accompanies TDL, and provides a variety of intermediate representations, analysis techniques, and graph transformations that are useful for working with dataflow graphs that have been captured by TDL. Mocgraph is a companion tool that is provided along with TDP. Mocgraph can be viewed as a library of algorithms and representations for working with generic graphs, whereas TDP is a specialized package for working with dataflow graphs.

For example, TDP includes a transformation tool to convert SDF representations

into equivalent homogeneous SDF (HSDF) representations, based on the transformation algorithm introduced in [14]. Such a transformation can in general expose additional concurrency that is not represented explicitly in the original SDF graph. In this chapter, we make use of both generic-graph-based (via Mocgraph) and model-based (via TDP) analysis methods to identify SSRs within CAL networks As we will demonstrate later in this chapter, automated identification of SSRs from CAL networks provides a powerful and novel methodology for optimized implementation of dataflow graphs. This methodology is especially useful in the design and implementations of embedded multiprocessors for video processing. In section 4.4.3, we develop the concept of SSRs in details.

Compared to other design tools for representation and transformation of dataflow graphs — such as SysteMoC [23], PeaCE [24], and stream-based functions [67] — a distinguishing feature of TDP is its support for representing and manipulating different specialized forms of dataflow semantics. This arises from the emphasis in TDL on recognizing a wide variety of important forms of dataflow semantics along with relevant modeling details that are required to meaningfully analyze those semantics. Due to this feature of TDP, its capabilities are highly complementary to those of existing dataflow-based frameworks, since TDL and TDP can be used to capture and analyze, respectively, representations from many of these frameworks.

4.4.2 Interface between DIF and CAL

Our method to optimize implementation of DSP applications combines the advantages of three complementary tools, as shown in Figure 4.5. The given DSP application is

initially described as a CAL network, which is a highly expressive form of dataflow graph. The CAL-based dataflow representation is then translated into a DIF-based intermediate representation for analysis by TDP. This TDP-driven analysis produces a set of SSRs, and an associated quasi-static schedule, which is then translated into a reformulated CAL specification. This transformed CAL code is then translated to a C code implementation using CAL2C. The generated CAL2C implementation is optimized to exploit the static structures provided by the SSRs and their enclosing quasi-static schedules.

In our current work, TDP reads XML representations of CAL actors and CAL networks, and then generates a TDL file based on the extracted information. We are also developing an interface between XML and TDL, through which TDL files can be represented in XML format, thereby making XML a bridge for communicating between different dataflow languages in our targeted CAL- and DIF-based design flow.

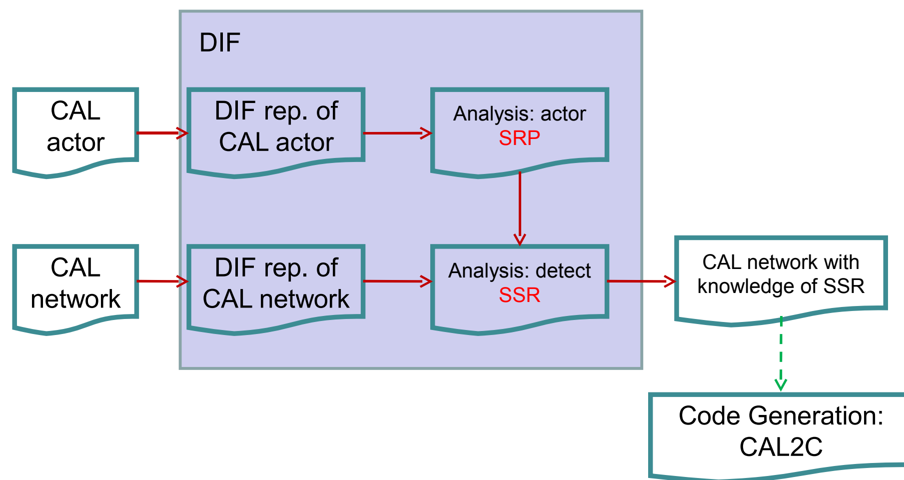


Figure 4.5: Overview of our CAL- and DIF-based method for optimizing dataflow graph implementation. SRP represents statically related port and SSR represents statically related region.

Describing an actor in CAL involves describing not only its ports, but also the structure of its internal state; the actions it can perform; what these actions do (such as token

production and token consumption, and updating of actor state); and how to determine the action that the actor will perform next. When performing network dataflow analysis, we analyze interactions among ports, state variables, and guard conditions of CAL actors. In our current research, which focuses on deriving and utilizing information about the token production and consumption rates of actors, action priority is not taken into consideration. This is because action priority only affects the order of action execution within individual actors; it does not affect the numbers of tokens that are produced or consumed.

4.4.3 Statically schedulable regions

Using TDP, one is able to automatically process regions that are extracted from the original network, and exhibit properties similar to synchronous dataflow (SDF) [14] graphs. SDF is geared towards *static scheduling* of computational modules, which can provide significant improvements in system performance and predictability for DSP applications. Detection of SDF-like regions is an important step for applying static scheduling techniques within a dynamic dataflow framework. Segmenting a system into SDF-like regions also allows us to explore another kind of intrinsic concurrency — that resulting from the dynamic dependencies between different regions. Using SDF-like region detection as a preprocessing step to software synthesis generally reduces the number of threads, and is well suited for efficient parallel implementation of video processing systems. In this chapter, we designed and implemented the *statically schedulable region detection algorithm* as part of TDP to address inter-actor concurrency.

Given a dataflow graph G consisting of CAL actors, one can construct a *port con-*

nectivity graph (PCG) $P = (V, E)$, where V, the vertex set of the graph, is the set of all ports of all actors in G, and E is a set of undirected edges. If there is an edge between a pair of ports$(A.a, B.b)$, the relationship between ports $A.a$ and $B.b$ satisfies two conditions: connectivity and statically-related numbers of tokens. When discussing a graphical representation of a CAL network, we assume that the representation is in the form of a PCG, unless otherwise stated.

Our approach for deriving statically schedulable regions involves partitioning and grouping actor ports based on relationships that pertain to various kinds of interactions between ports.

This overall process of partitioning and grouping begins at the level of individual actors. Ports inside an actor can be viewed as having different kinds of associations with one another. Some ports can be viewed as related because they are involved in the same action, while some are related because they affect the same state variable. We refer to the set of ports in A as the port set of A, denoted as $ports(A)$. For a given action $l \in \Gamma(A)$, the set of ports that can be affected by the action is denoted (allowing a minor abuse of notation) by $ports(A)_l$. In this chapter, we apply the following two kinds of port associations:

1. $\exists(l \in \Gamma(A))$ such that $a, b \in ports(A)_l$;

2. $\exists l, m \in \Gamma(A)$ such that $a \in ports(A)_l$, $b \in ports(A)_m$, l is a state-changing action, and m is a state-guarded action.

We define these two conditions as the *coupling relationships*, and we observe that in general, two distinct ports can satisfy zero, one or both of the coupling relationships.

In the case that one or both of the coupling relationships are satisfied, we say that these two ports have strong connections.

As shown in Figure 4.6, there are four stages in our application of the PCG: coupled ports (CPs), coupled groups (CGs), statically related groups (SRGs), and statically schedulable regions (SSRs). Using TDP, we repeatedly apply two key techniques when working with the PCG — techniques of partitioning and grouping — through the connected component analysis of the PCG. Transformation of PCG is the procedure of all the ports in the CAL network going through the above four stages. The detailed description on strong connections, statically schedulable regions and PCG derived in our design flow is the result of network analysis in TDP [43].

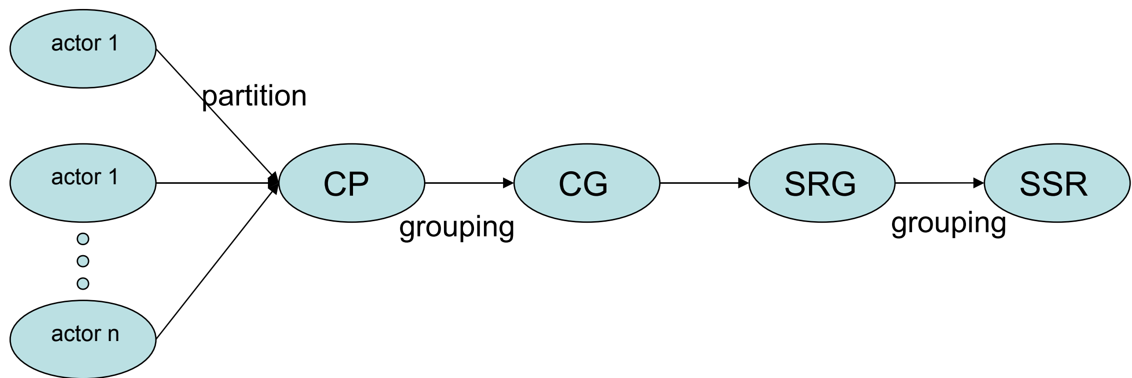


Figure 4.6: SSR detection in PCG.

By transforming the PCG for a CAL network, we obtain a set of SSRs. In general, this set can be empty or it can contain one or multiple elements. For individual actors, SSRs distinguish "strong" connections from "weak" connection among ports in terms of static schedule-ability analysis. Regarding the CAL network, SSRs combine parts of the system that exhibit potential for efficient static or quasi-static scheduling.

4.4.4 Mapping SSRs into multi-core systems

CAL provides for effective concurrent programming, which provides natural benefits for multi-core systems. However in the available code generators for CAL, such as CAL2C, no optimization is performed for CAL actors. SSRs distinguish weak connections from strong connections among ports. Each SSR is grouped and subsequently applied as a thread to help optimize the multi-threaded implementation for a multi-core target. The main differences between SSR-based threads and CAL-actor-based threads lie in two aspects: On one hand, each SSR-based thread can be quasi-statically scheduled, which allows for significant compile-time streamlining of the associated scheduling mechanisms. On the other hand, data connections between SSR-based threads are much weaker compared to intra-SSR connections. This latter property improves interprocessor communication. For these reasons, SSRs provide enhanced granularity for parallelization on multi-core systems.

Figure 4.7 illustrates SSRs within the IDCT subsystem. Here, the main body of the IDCT is composed of the actors *row*, *tran*, *col*, *retran* and *clip*. The *dataGen* and *print* actors are used to complete a testbench for the network — $dataGen$ is responsible for generating input data, and $print$ for displaying the output from the IDCT computation. The shaded regions shown in the figure correspond to the different SSRs, which are unique to the application.

Next, we consider mapping of SSRs into multi-core systems. If we temporarily ignore the load balancing of computational tasks, we map one SSR into one core. In the example of the IDCT subsystem, there are two SSRs, which can be mapped naturally for

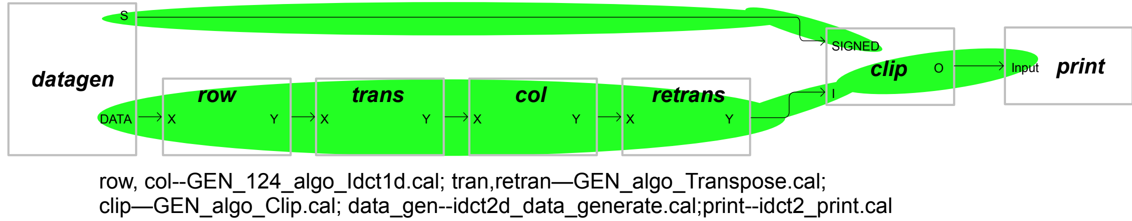


Figure 4.7: SSRs in the IDCT subsystem.

a dual-core system. If all of the ports in one actor belong to the same SSR, we allocate the actor onto one core. On the other hand, for an actor that has ports belonging to different SSRs, we divide the actor into two or more parts, and each part is allocated separately — thus, in general, actors may be "split" across multiple cores if they are separated by the SSR construction process. As we described before, SSRs distinguish strongly related ports from relatively weaker connections. For example, inside one actor, two SRGs may interact with one another only through processing of shared state variables. The mechanism to access such shared data can be easily implemented in a multi-core system, such as through use of semaphore primitives.

In another word, SSR distinguish weak connections from strong connections. Thus, when two SSRs are allocated onto two cores, the connections for the SSRs between the cores are weak. In our example, semaphores can be used for the two cores to access the same data. In an SSR-based multi-threaded system, data movement between cores is reduced, and it takes correspondingly less time and effort for memory management and synchronization between cores. In this sense, SSR-based systems are effective in exploiting data locality for multi-core systems.

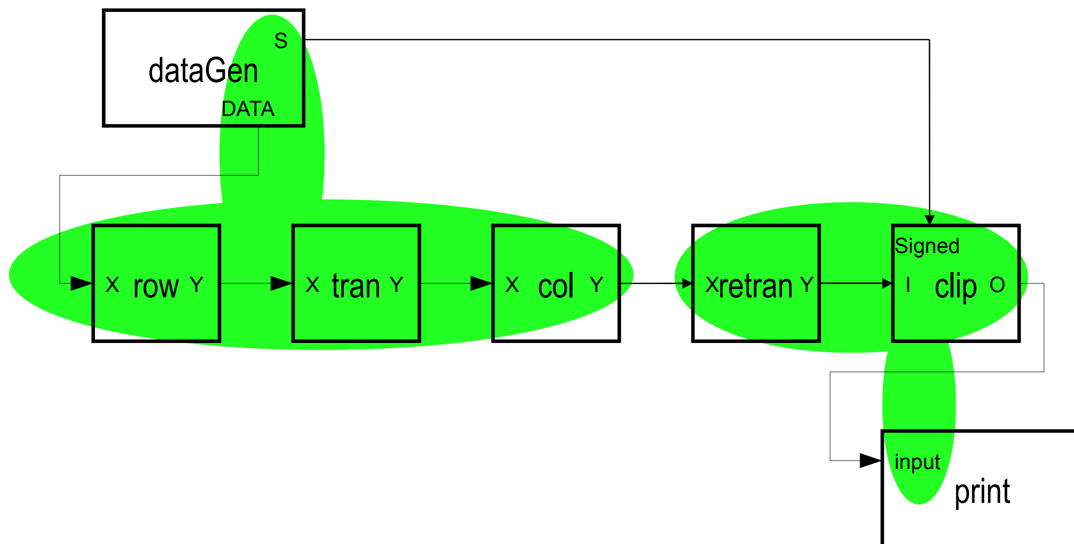
DMA is helpful for intra-chip data transfer in our implementation on multi-core processors, where each processing element is equipped with a local memory and DMA

is used for transferring data between the local memory and the main memory. Multi-core systems that have DMA channels can transfer data to and from devices with significantly less CPU overhead. Similarly, a processing element inside a multi-core processor can transfer data to and from its local memory without occupying its processor time, which provides for computation and data transfer concurrency. Using DMA, data communications between actors are concurrent with the computations, and therefore concurrency can be further enhanced. Adapting DMA into our hardware platform is a promising direction of future research.

Each SSR can be scheduled quasi-statically, which means a significant portion of the schedule structure can be fixed at compile time. Scheduling of each SSR can be controlled in the core allocated for the SSR. Scheduling control is centralized regarding synchronization between SSRs. For two SSRs that share data, the central scheduler must determine the order of execution between the SSRs.

Suppose that we have a dual-core platform. If we map the tasks based on actors, as implemented in the original CAL2C, one option is shown in Figure 4.8. Four CAL actors are mapped into one core, and the other three actor are mapped into the other core. There are other possible options with differences in the numbers of actors that are mapped to individual cores. Whatever option is used for mapping actors, although inter-actor concurrency is maintained, for each macroblock processed by the IDCT module, execution of actors is sequential. Furthermore, since there are two paths between actors *dataGen* and *clip*, as shown in Figure 4.8, if these two actors are mapped onto separate cores, there is a relatively large amount of data communication between the cores, which in turn results in a large amount of context switch overhead on the individual cores.

If we map the IDCT onto a dual-core system based on SSR analysis, a straightforward mapping for this case is shown in Figure 4.7. In this case, the connections between the cores are weak connections inside both the *dataGen* and *clip* actors. These weak connections can be implemented using semaphore primitives. Furthermore, inside each core, the actions can be statically scheduled in terms of checks on an appropriately defined semaphore. Here we can easily take advantage of well known SDF scheduling techniques, such as APGAN [68]. An example of scheduling of SSRs, including the actor *clip*, is shown in Figure 4.9.



row, col--GEN_124_algo_Idct1d.cal; tran, retran--GEN_algo_Transpose.cal;
clip--GEN_algo_Clip.cal; data_gen-- idct2d_data_generate.cal; print-- idct2d_print.cal;

Figure 4.8: Actor-level mapping onto a multi-core platform.

After integrating results of SSR analysis into CAL2C, we obtained a modified version of CAL2C, which we call *CAL2C-SSR*. To evaluate the effectiveness of our SSR techniques, we conducted experiments on a dual-core 2.5Ghz laptop. We generated C

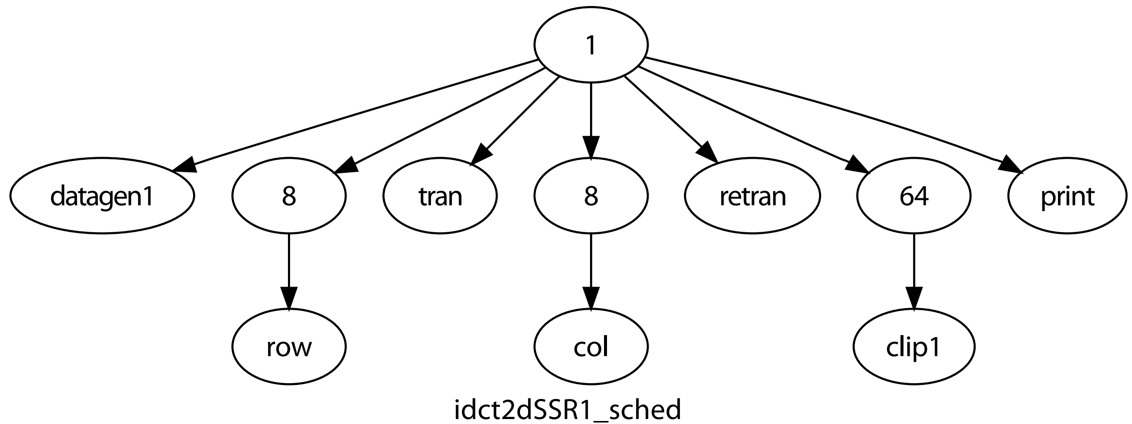


Figure 4.9: Scheduling tree for one SSR in the IDCT.

code using CAL2C and CAL2C-SSR for three different *IDCT* versions. The first version (V1) does not employ any SSR analysis, and can be viewed as being scheduled purely through SystemC, which is used in CAL2C. In this version, the actors are mapped onto two core as shown in Figure 4.8.

The second version (V2) uses CAL2C-SSR. This version exploits the SSRs illustrated in Figure 4.7, and employs a quasi-static integration of static schedules for these SSRs with top-level dynamic scheduling. In this version, two SSRs are mapped onto two cores, and semaphore primitives are used for inter-SSR communication.

The third version (V3) also uses CAL2C-SSR. This version also uses a modified, more predictable version of the *clip* actor that can be used when the input data is known in advance. In the new version of *clip*, the ports *Signed* and *O* are rewritten to become coupled ports. Then the original two SSRs are combined as one SSR through connections inside *clip*. In the illustration of V3 shown in Figure 4.10, the *IDCT* system becomes an SDF model that runs as a single thread. Since entirely static scheduling is used in this version, V3 is the most efficient in terms of execution speed.

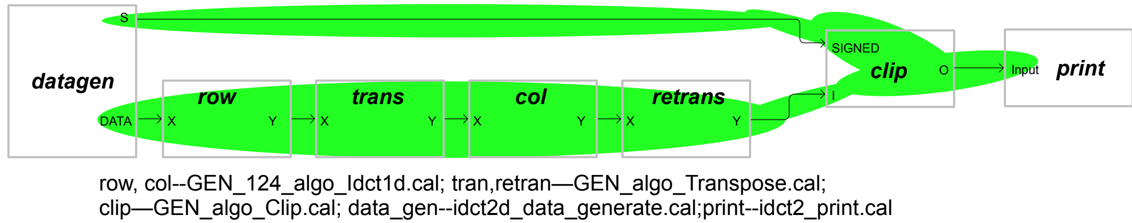


Figure 4.10: IDCT subsystem with one SSR.

We experimented with all three *IDCT* versions using Microsoft Visual Studio. The results are shown in Figure 4.11. Here, V2 shows an improvement in performance of 1.5 times compared to V1, whereas V3 shows the best performance among all three versions.

Note that while V3 exhibits the best performance, demonstrates that larger SSR regions can lead to significant improvements in performance, and is generally interesting as a kind of "limit study", this version is not of practical utility. This is because V3 requires prior knowledge of input data, which is not a practical assumption for real-time operations.

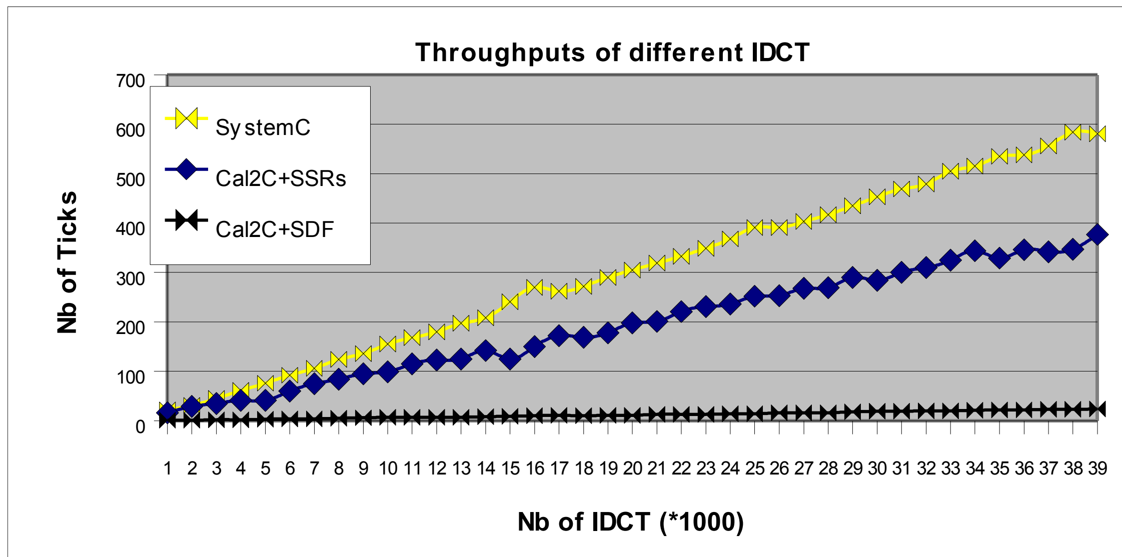


Figure 4.11: Results: clock cycles vs number of iterations.

4.4.5 Concurrency analysis of the MPEG-4 SP decoder

When we analyze the MPEG-4 SP decoder in Figure 4.2 in the domain of TDP, the first step is to translate the hierarchical system into a flattened one in which every actor is an atomic actor.

In TDP, a *fork* actor is introduced to implement dataflow-style broadcasting when needed (i.e., when data must be copied to multiple outgoing edges). For example, *Header* is an atomic actor inside the super actor *parser* in the CAL network of Figure 4.2, and the tokens produced from the *BTYPE* port of the actor *Header* are broadcast to five different input ports of different actors. Thus, in the intermediate representation derived by TDP, a fork actor *GEN-mgnt-fork* is inserted between *Header* and the five actors that are destinations of the broadcast. Conceptually, whenever *GEN-mgnt-fork* fires, it consumes a single token and produces copies of that token onto its five output ports. Due to space limitations, the PCG graph of the MPEG RVC decoder is not illustrated in this chapter.

When applied to the targeted decoder system, our tools for SSR detection return a total of 30 SSRs that are detected. Each SSR can be statically scheduled in terms of some enclosing condition. Since SSRs can be processed concurrently, the SSRs become the basic unit for thread formation instead of actors. Compared with actor-based threads, SSR-based threads provide advantages such as reduced inter processor communication (IPC) and synchronization overhead between threads. These advantages are important since IPC and synchronization overhead are often limiting factors for performance enhancement in multi-core platforms.

We further modified the scheduler of CAL2C to better accommodate SSRs [46].

MPEG-4 SP decoder speed	frame/second
monoprocessor with systemC scheduler	8
monoprocessor with round robin scheduler	42
monoprocessor with round robin scheduler and SSR	44
dual-core processor with round robin scheduler and SSR	50

Table 4.1: MPEG-4 SP decoder performance.

All of the SystemC primitives have been removed from the current version of Cal2C. The current scheduler of CAL2C is improved into a round robin scheduler [69] executing each actor in a loop; an actor is fired until input tokens are available and output FIFOs are not full. SSRs can be easily incorporated in this fully software-based implementation, independent from SystemC, by removing all of the possible tests on the FIFOs when an SSR is detected.

We conducted experiments involving the application of CIF sequences with size 352x288. A CIF-size image (352x288) corresponds to 22x18 macroblocks. As shown in Table 4.1, the experimental results demonstrate that CAL2C with SSR on the round robin scheduler has the best performance in a multi-core system.

Note that although we have detected many SSRs in the whole MPEG-4 SP decoder system, we have applied SSRs only to three parts within the IDCT system. These are parts where SSR detection has significant impact. A completely thorough application of SSRs would require much more effort, but we expect that such an effort would result in further improvements. This is a useful direction for further exploration in this case study.

We relate the number of ports in one SSR to the scale of the SSR granularity due to the general fact that a larger number of ports result in a bigger sequence of actions. In some cases, however, SSRs may produce too large a granularity to promote effective computational load balancing. In such cases, further dataflow analysis techniques are

needed to decompose “large” SSRs into smaller units that are more computationally-balanced. Similarly, it may be advantageous to combine fine-grained (“small”) SSRs into larger units to further promote the streamlining of IPC and synchronization. Thus, SSR detection provides an important step towards improving the dataflow granularity of CAL programs; however, there may be room for significant further improvement through post-processing transformations that operate on the detected SSRs. Some work along these lines has already been developed as part of the PREESM project [70]. Further exploration on this class of “granularity-adjustment” transformations for SSRs is a useful direction for further work.

Chapter 5

Automatic Integration of SSR and Cal2C

5.1 overview

In this chapter, we apply the proposed framework to video processing systems. We propose how to represent DIF language structures in an XML format, which assists automation and integration for efficient system generation, as shown as red, bold and italic in Figure 5.1.

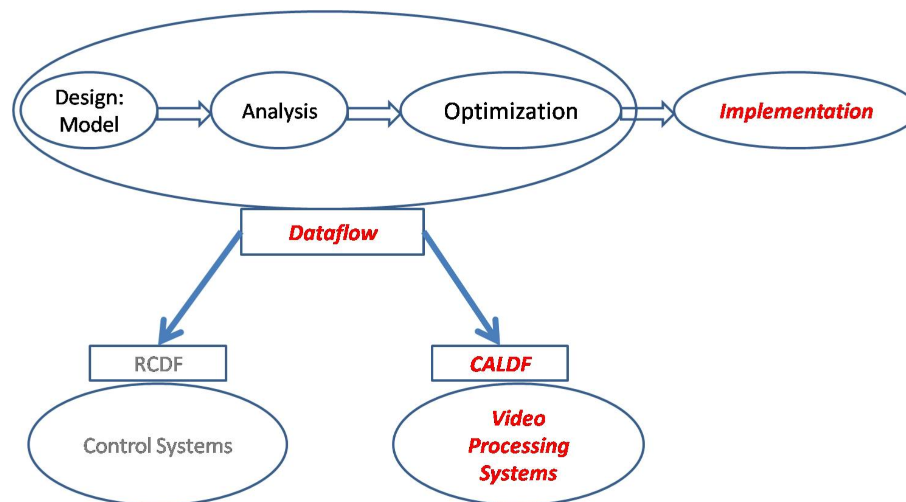


Figure 5.1: Framework for video processing systems: Implementation.

Upcoming MPEG video coding standards are intended to increase the quality and

flexibility of complex and versatile future video coding applications. Since 1988, several MPEG standards have been developed successfully based on available hardware technologies and software support. Early MPEG standards (MPEG-1 and MPEG-2) were specified by textual natural-language descriptions. Starting with MPEG-4, reference software written in C/C++ became the formal specification of the standard. Written in a sequential programming language, this reference software describes a sequential algorithm, effectively hiding the considerable inherent concurrency of a video decoder. Furthermore, the reliance on global memory and state makes the reference description difficult to modularize, resulting in a very monolithic specification format.

At the same time, multi-core devices, which incorporate two or more processors on the same integrated circuits, are becoming increasingly relevant to the design and implementation of DSP systems (e.g., see [63]). Efficient deployment of video processing applications on multi-core systems requires effective parallel exploitation of task level concurrency in order to improve system performance. The drawbacks of existing video standard specification formats and the increasing importance of multi-core platform technologies motivated the development of the Reconfigurable Video Coding (RVC) standard [48]. The key concept of RVC is to enable design and specification of decoders at a higher level of abstraction than that provided by generic, monolithic C-based specifications, and improve high level application analysis and optimization, including exploitation of parallel processing resources.

Dataflow-based programming, with its intrinsic concurrency, is employed in a wide variety of commercial and research-oriented tools related to DSP system design. Dataflow modeling techniques underlie many popular graphical tools for digital signal processing

(DSP) system design (e.g., see [16]). In DSP-oriented dataflow graphs, vertices (*actors*) represent computations of arbitrary complexity, and an edge represents the flow of data as values are passed from the output of one computation to the input of another. A variety of dataflow-based languages and tools have been developed for design and implementation of embedded DSP systems. Although all of these languages share the property of data-driven communication between actors, distinct languages generally differ in terms of specialized dataflow modeling features and associated support for analysis and optimization techniques.

Synchronous dataflow (SDF) is a specialized form of dataflow that is streamlined for efficient representation of DSP systems [14]. SDF is a restricted model that handles a limited sub-class of DSP applications, but in exchange for this limited expressive power, SDF provides increased potential for static (compile-time) optimization of DSP hardware and software (e.g., see [44]).

A number of dataflow-based formalisms have been developed to describe applications that involve dynamic dataflow behavior. For example, CAL [1] is a language for specifying dataflow actors in a way that is fully general (in terms of expressive power), while clearly exposing functional structures that are useful in detecting important special cases of actor behaviors (e.g., SDF or SDF-like actor behaviors). The CAL language, in terms of its high level of abstraction, is similar to the Stream-Based Functions (SBF) model of computation [19]. Both models share common features relating to modeling of dynamic dataflow behaviors. However, SBF combines the semantics of both dataflow models and process network models, while CAL extends the dataflow model by enriching the properties of individual actors. Furthermore, CAL is a fully-featured programming

language, providing both an abstract, dataflow model of computation as well as a comprehensive set of operators and other semantic features for completely specifying the internal behavior of dataflow components.

The DIF language (TDL) provides a standard approach for specifying DSP-oriented dataflow graphs at a high level of abstraction that is suitable for both programming and interchange (across different dataflow-based languages or tools) [8]. TDL provides a unique set of semantic features for specifying graph topologies, hierarchical design structure, dataflow-related design properties, and actor-specific information. TDP (The DIF Package) accompanies TDL, and provides a variety of intermediate representations, analysis techniques, and graph transformations that are useful for working with dataflow graphs that have been captured by TDL.

In order facilitate integration of TDL and TDP into design flows, we present in this chapter a common XML-based format called *DIFML*. DIFML is designed for structured exchange of design information between different dataflow-based specification formats, such as TDL and CAL.

In previous work, we have formulated systematic SSR detection and implemented SSR region detection using TDP (The DIF Package) [71]. Code generation from CAL to C (CAL2C) has also been developed in previous work [3], and we have explored integrated application of CAL, TDP, and CAL2C using manual techniques [71]. Simulation results from such manual integration demonstrated that the integrated application leads to improved exploitation of parallelism [22].

This chapter builds on these previous efforts, and presents an automated approach for integrating SSR derivation into implementations that are synthesized from CAL spec-

ifications. To facilitate such an automated and integrated design flow, we also present in this chapter a new XML-based format, called DIFML, which we have developed to exchange information between different dataflow tools. We present experimental results on a reconfigurable video coding application to demonstrate the effectiveness of our automated toolset.

5.2 Related work

5.2.1 Design Flow

Embedded system design and implementation can be a time-consuming process requiring intensive effort, resources, and time. Hardware description languages (HDLs), such as Verilog HDL [72], are widely used in the design of embedded systems. In an attempt to reduce the complexity of designing in HDLs, which have been compared to the equivalent of assembly languages, a variety of efforts have emerged to raise the abstraction level of associated design processes. For example, companies such as Cadence, Synopsys and Agility Design Solutions are promoting SystemC as a way to combine high level languages with concurrency models to allow faster design cycles for FPGAs than is possible using traditional HDLs.

Approaches based on standard C or C++ (with libraries or other extensions allowing parallel programming) are found in the Catapult C tools from Mentor Graphics, and in the Impulse C tools from Impulse Accelerated Technologies. Languages such as SystemVerilog [73] seek to accomplish the same goal, but are aimed at making hardware engineers

more productive versus making FPGAs more accessible to software engineers.

There are also a number of high level languages targeting embedded systems. For example, StreamIt [74] is a programming language for high-performance streaming applications. Annapolis Micro Systems, Inc.'s CoreFire Design Suite and National Instruments LabVIEW FPGA provide a graphical dataflow approach to high-level design entry. Our work in this chapter is related to such efforts in being tightly coupled with CAL, which is a language oriented towards design and implementation of embedded systems from a high level of abstraction. In addition to the coupling with CAL, another distinguishing aspect of our work is its focus on the domain of video processing, and in particular, reconfigurable video coding.

5.2.2 XML format

The extensible markup language, widely known as XML, is a markup language that was created by the World Wide Web Consortium (W3C) to overcome limitations of HTML. Like HTML, XML is based on SGML — the Standard Generalized Markup Language. Although SGML has been used in the publishing industry for decades, its perceived complexity intimidated many people that otherwise might have used it. XML was designed with the Web in mind.

A major advantage of XML is that one can encode document information more precisely compared to HTML. This means that programs processing these documents can "understand" them much better and therefore process the information in ways that are not possible for ordinary text processors.

One major application of XML is to make web pages with decent layout that are universally accessible, regardless of browser type. XML also lets one check whether or not optional features are present, and allows for invocation of alternative code to take care of cases where such features are missing.

XML is a promising candidate for carrying data associated with high level text based languages for subsequent use. XML itself is designed to be self-descriptive, which ensures that all of the information from the original file can be understood by other applications. XML tags are not predefined by users. It can be convenient for users to design appropriate tags to describe the context of the information being exchanged.

Representing different languages using a common XML format allows for integrated use of heterogeneous languages within a design flow, thereby allowing designers to combine the unique strengths and features associated with different languages. In our work, as shown in Figure 5.2, we use CAL to design the targeted system, DIF to optimize the system, and Cal2C as a back-end implementation process. The interfaces in our design flow between CAL and DIF, and between DIF and Cal2C, are based on CALML (an XML-based format associated with CAL), and DIFML, respectively.

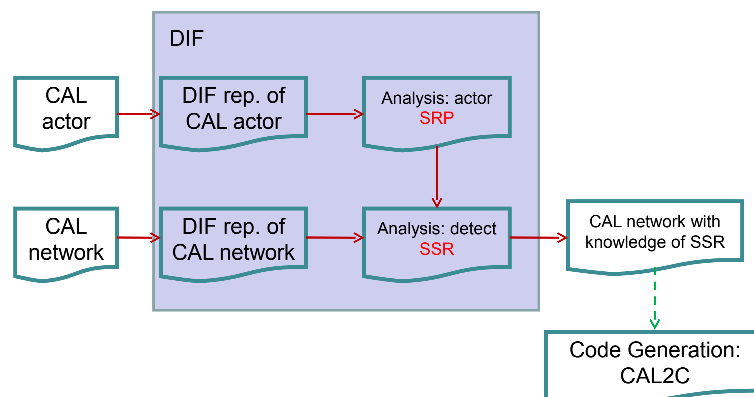


Figure 5.2: Automation of efficient video processing system generation.

5.3 Automated Approach

Our proposed design-to-implementation process is illustrated in Figure 5.3. Here, CAL is used to describe and model the functionality of the targeted system. DIF and TDP are then applied for analysis and exploration of optimization alternatives. Different optimization techniques target different performance measures, such as real time constraints, power consumption, or buffer size. In this chapter, we focus on optimizing the execution speed of the targeted RVC systems.

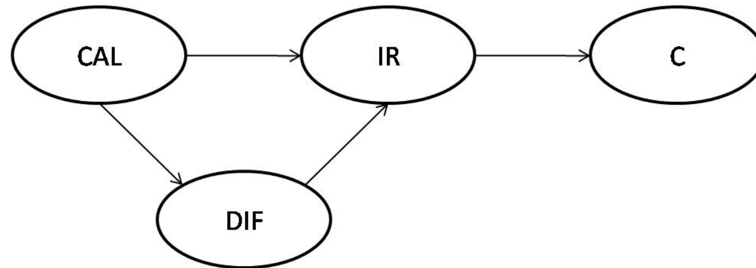


Figure 5.3: Automated design-to-implementation flow.

The Open RVC-CAL compiler (Orcc) [47] is a tool set under BSD license to realize the automated design-to-implementation flow for the RVC-CAL dataflow programming language. It has been developed with a back-end that performs CAL-to-C transformation. Transformation to other lower level languages, such as Java, is under development.

The input to the Orcc is an application that is in terms of CAL actors and a CAL network. CAL actors are represented in the form of .cal files, and the CAL network is specified as .xdf file. The output is an automatically generated implementation, which is targeted to a lower level language, such as C, C++ or Java.

The compiler is divided into two phases — a front end and a back end. The front

end is responsible for parsing actors and networks, flattening the hierarchical network, and generating actors in the JASON format. The back end is responsible for generating an implementation in a user-specified lower level language.

5.3.1 Intermediate Representation

The Intermediate Representation (IR) used in Orcc is managed in the form of .jason files.

The top-level structure in the Intermediate Representation is an actor. An actor contains:

- parameters
- input ports
- output ports
- state variables
- a list of functions/procedures
- a list of actions
- an action scheduler

A variable is represented by the Variable class. A Variable has a location, which is the place in the source file where it was declared, a type, a name, and the list of its uses. The list of uses (called ‘def-use‘) is automatically computed and maintained by Orcc.

A Variable also has two attributes that may be used depending on the context: a Variable may have an initial expression, with the exception of local variables, and a Variable may have a value, which is its runtime value. The value of a Variable is only used when an actor is interpreted.

A GlobalVariable is a Variable whose initial expression may be evaluated as a constant, and accessed with the getConstantValue method.

A StateVariable is a GlobalVariable that has an additional "assignable" attribute. This attribute records the information about whether a variable can be assigned or not.

A LocalVariable is a Variable that has an "assignable" attribute (like a StateVariable), an SSA (static single assignment) index, and an "instruction" attribute. The "instruction" attribute references the assign instruction where the variable is assigned for the first and only time.

A procedure has parameters and local variables. It has a body made of a list of CFG nodes. A CFG node corresponds to a node in the Control Flow Graph, and is defined by the interface. There are three types of nodes: a BlockNode, an IfNode, and a WhileNode.

Scheduling information (priorities and FSM) are present in the action scheduler. Actions are sorted by descending priority, so the action with the highest priority comes first.

5.3.2 Integrating Results of DIF Analysis into the C Back End

In our targeted design flow, the analysis of CAL networks and CAL actors is conducted in the DIF environment, as shown in Figure 5.2. In our current implementation,

we detect statically schedulable regions (SSRs) from the DIF-based analysis to optimize scheduling structures for efficient implementation. Currently the input to this form of DIF analysis is a CAL network along with its constituent CAL actors. The output is a set of SSRs, and static schedules corresponding to those SSRs. This SSR and schedule information is generated for efficient system implementation.

The back end of the code generator adopts a round-robin scheduling approach. Round-robin (RR) is a simple scheduling algorithm for executing multiple tasks in an operating system. In the form of RR scheduling that we apply, time slices are assigned to each task in equal portions and in circular order, and no priority ordering is considered across the tasks. Round-robin scheduling is simple, easy to implement, and starvation-free. In the generated system, there is a main scheduler that takes care of all actor schedulers. The main scheduler passes the right of execution to the actor schedulers one by one. When an actor scheduler is selected for execution, and dataflow requirements for one or more actions within the actor are satisfied, the actor scheduler will execute an appropriate action. Then the right of execution is passed to the next actor scheduler.

Static scheduling can be integrated into the RR scheduler in the following way. If some actors can be statically scheduled, that is, the execution of some actors is determined to be continuous and fixed in compile time, then we can combine the schedulers of these actors into one scheduler. For example, suppose row and $transpose$ are actors and $row_scheduler$ and $transpose_scheduler$ are corresponding schedulers. Based on SSR detection in TDP analysis, we can determine the scheduling of these two instances as always following the pattern shown in Figure 5.4. Thus, we can reduce the number of schedulers into one.

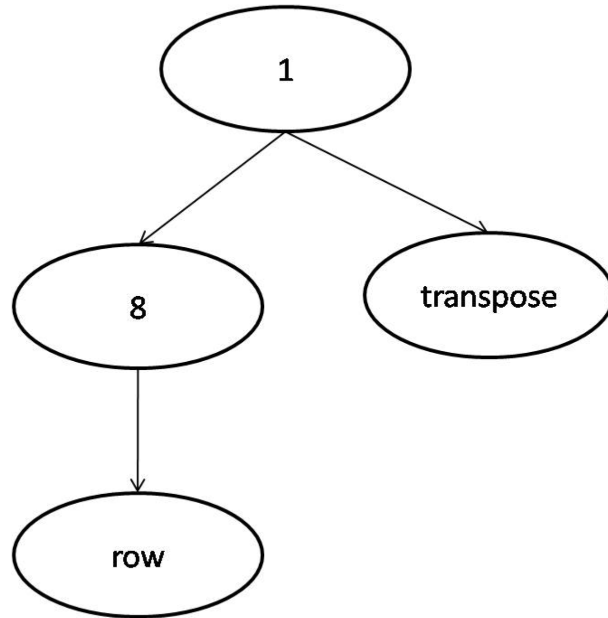


Figure 5.4: Static scheduling: actors row and $transpose$.

A number of related efforts are underway to develop efficient scheduling techniques for CAL networks. The approach of Platen and Eker [51] sketches a method to classify CAL actors into different dataflow classes for efficient scheduling. Boutellier et al. [52] propose an approach to quasi-static multiprocessor scheduling of CAL-based RVC applications. The approach involves the dynamic selection and execution of “piecewise static schedules” based on novel extensions of flow shop scheduling techniques.

Many previous research efforts have focused on task mapping for multiprocessor systems from other kinds of specification models or languages (e.g., see [16]). For example, Li et al. [53] provide a method for allocating and scheduling tasks using a hybrid combination of a genetic algorithm and ant colony optimization. The approach involves consideration of both global and local memory spaces across the targeted multiprocessor system. Ennals et al. [54] develop a method for partitioning tasks on multi-core network processors.

Compared to prior work on dataflow techniques and multiprocessor system design, major unique aspects of our approach for scheduling are the capability to decompose CAL actors based on their formal action- and port-based semantics, and to construct and subsequently transform SSRs and SSR actors from these decomposed representations.

When integrating SSRs into real implementations, we distinguish between two kinds of SSRs, as shown in Figure 5.5. In the first type, all CAL actors inside the SSR are preserved from their original structures in the corresponding CAL network, such as $SSR1$ in Figure 5.5. In the second kind of SSR, there is at least one partial CAL actor, of which some ports do not belong to the SSR, such as $SSR2$. When implementing SSRs of the second type, we divide each partial actor into two separate actors, as shown in Figure 5.6. In Figure 5.6, actor C is split into two new actors: C_1 and C_2. C_1 is statically scheduled in SSR_2, and C_2 has its own dynamic scheduler. Currently, implementation of the first kind of SSR is complete, and integration of the second kind of SSR is under development.

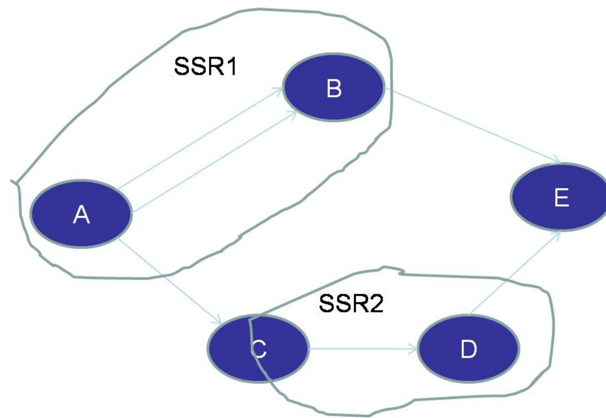


Figure 5.5: Two kinds of statically schedulable regions.

Figure 5.7 shows three kinds of options to integrate SSRs into Orcc. Option 3 is to modify the generated C code by integrating SSRs. Option 2 is to introduce SSRs into

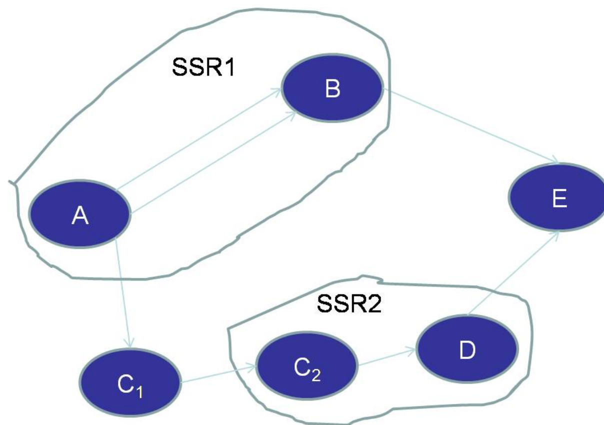


Figure 5.6: SSR: splitting one CAL actor into two actors.

intermediate representations, that is, into generated intermediate code that is based on .jason files and .difml files. Option 1 is to introduce SSRs in the front end where the CAL network is parsed into JASON files. Option 3 is generally the simplest to implement, while option 1 has the potential to produce more efficient implementations since the structure of SSRs can be exploited more rigorously in scheduling and related dataflow transformations.

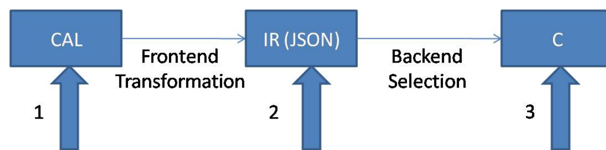


Figure 5.7: Code generation procedure.

We have implemented option 3 as an initial prototype of SSR integration. In our ongoing and future work, we are exploring implementations of options 2 and 3.

5.4 The DIFML format

As described previously, the dataflow interchange format (DIF) is proposed as a standard approach for specifying and integrating arbitrary dataflow-based semantics for DSP system design [8], and The DIF language (TDL) is an accompanying textual design language for high-level specification of signal-processing-oriented dataflow graphs.

In order to describe DIFML, we introduce a number of concepts associated with the general XML format: node, element, attribute and tags. A node is a part of the hierarchical structure that makes up an XML document. "Node" is a generic term that applies to any type of XML document object, including elements, attributes, comments, processing instructions, and plain text. A tag is a markup construct that begins with < and ends with >. Tags come in three flavors: start-tags, for example `</section>`, end-tags, for example `</section>`, and empty-element tags, for example `<line-break/>`. An element is a logical component of a document. An element either begins with a start-tag and ends with a matching end-tag, or consists only of an empty-element tag. The characters between the start- and end-tags, if any, are the element's content, and may contain markup, including other elements, which are called "child elements". An attribute is a markup construct consisting of a name/value pair that exists within a start-tag or empty-element tag.

DIFML is designed as an XML-based format for exchanging information between TDL and other tools and languages, and more generally, between arbitrary pairs of dataflow environments. There are different elements in DIFML and these elements are listed in a

hierarchical way. The element at the highest level is *graph*, while *topology* and *interface* are lower level element. Under *topology*, there are three elements at the same level: *nodes*, *edges* and *interface*. For each element, there are three kinds of attributes: *implicitAttributes*, *builtInAttributes* and *userDefinedAttributes*. ImplicitAttributes are those attributes necessary and inherent to the element, such as the id of a node. BuiltInAttributes are attributes that are recognized as part of the DIF language, typically through corresponding reserved words or other kinds of language constructs. For example, for an edge element in and SDF model within a DIF graph (i.e., within a graph that is defined with the SDF keyword), there are three kinds of builtInAttributes: the production rate, consumption rate, and delay. UserDefinedAttributes are attributes that users add to selected elements at their own discretion. The following is a simple example of an SDF model in the DIFML format. For conciseness, we just show part of the associated DIFML file.

```
<?xml version='1.0' encoding='UTF-8'?>
<difml xmlns='http://www.ece.umd.edu/DIFML'>
    <graph>
        <implicitAttributes>
            <name val='dat2cd'/>
            <type val='SDFGraph'/>
        </implicitAttributes>
        <topology>
            <nodes>
                <node>
                    <implicitAttributes>
                        <id val='A'/>
                    </implicitAttributes>
```

```
                <builtInAttributes>
                    <nodeWeight type='DIFNodeWeight'/>
                </builtInAttributes>
                <userDefinedAttributes>
                    <attribute name='output' type='Edge' val='e1'/>
                    <attribute name='readerFP' type='DIFParameter' val='reader'/>
                </userDefinedAttributes>
            </node>
        </nodes>
        <edges>
            <edge>
                <implicitAttributes>
                    <id val='e1'/>
                    <sourceId val='A'/>
                    <sinkId val='B'/>
                </implicitAttributes>
                <builtInAttributes>
                    <edgeWeight comsumption='[2]' delay='0'
                        production='[1]' type='SDFEdgeWeight'/>
                </builtInAttributes>
            </edge>
        </edges>
    </topology>
    <interface>
        <port>
            <implicitAttributes>
                <direction id='InA' nodeId='A' val='IN'/>
            </implicitAttributes>
        </port>
        <port>
            <implicitAttributes>
                <direction id='OutE' nodeId='E' val='OUT'/>
            </implicitAttributes>
        </port>
```

```
        </interface>
    </graph>
    <!--Automatically generated from DIF file-->
</difml>
```

As shown in the above example, each DIFML element contains an opening tag, a closing tag, and some content. The opening tag begins with a left angle bracket (<), followed by an element name that contains letters and numbers (but no spaces), and finishes with a right angle bracket (>). Following the content is the closing tag, which exhibits the same spelling and capitalization as the opening tag, but with one small change: a / appears right before the element name. Note that there is an element named *node*. This name is in correspondence with the related definition in the DIF language, and has different meaning with the node concept in XML terminology, which is a generic concept that applies to any type of XML document object.

Currently, the DIFML parser supports several major dataflow models that are recognized in the DIF language, including SDF [14], cyclo-static dataflow (CFDF) [17], parameterized synchronous dataflow (PSDF) [75], CAL dataflow (CALDF) [1], and multidimensional synchronous dataflow (MDSDF) [76].

5.5 Experimental results

We apply our automated design-to-implementation flow to an RVC MPEG4 SP decoder. We generate three kinds of code using Orcc tools:

1. C code with "C" as back end;

2. C code with "C+SSR" (C code generation integrated with derived SSRs) as back end;

3. C code with "C+modified_SSR" — based on the derived SSRs, we manually compute token production rate and consumption rate information to enhance static scheduling.

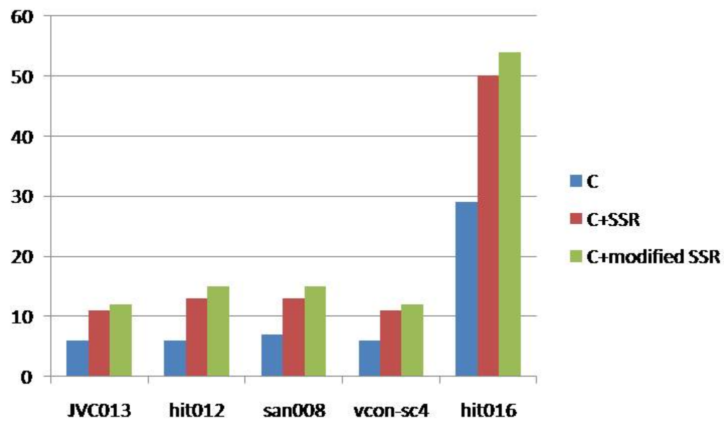


Figure 5.8: Experimental results for MPEG4 RVC SP decoder.

We generate three kinds of C projects using CMake. The projects are compiled and built using Microsoft Visual C++ 2008. The generated executables are executed on a Sony VAIO laptop with an Intel Pentium 1.2GHz processor. The experimental results are shown in Figure 5.8.

The results show approximately a factor of two improvement in performance after we integrate SSR derivation and scheduling. If we modify the CAL actors based on results of SSR derivation, the performance is even better. Currently, the modification based on SSR is performed by hand. Automating this part of the optimization process is an interesting direction for future work.

The results show a significantly higher frame rate on benchmark 5 of hit016. This is because for the first four benchmarks, the display sequence is set to 352x288, while for the fifth benchmark the display sequence is 176x144. The smaller display size consumes less resources and runs faster.

Chapter 6

Summary and Future Work

For embedded systems, dataflow provides powerful tools for modeling applications, and analyzing properties of hardware and software implementations. This proposal explores the use of parallelism to improve system performance in embedded systems. Because our methods improve both performance and predictability, the research has important applications in real-time systems, where performance constraints must be met in a reliable way.

6.1 Framework for Fast Parallel Implementation of Model Predictive Control

In this thesis, we have proposed a general framework for modeling, analyzing, and developing fast parallel implementations of the algorithms used in model predictive control (MPC). We have illustrated the use of this approach by application to the Newton-KKT part of the computations for a practically important class of MPC problems. We have demonstrated in simulations that this approach does result in implementations of

MPC that require much less computing time.

Much remains to be done. Currently we only deal with convex problems. We also can apply the methodology to non-convex problems, which may have multiple local optimal solutions. By starting several different initial points using the same or different algorithms, it is possible to improve the performance of finding global optimal solutions. Furthermore, because the communication times are greatly dependent upon the specific hardware the methods described here need to be applied to different examples of hardware.

The full collection of MPC algorithms is much richer than those analyzed here. Many MPC algorithms are much more complicated and require much more time than the ones analyzed here. These techniques have the potential to greatly decrease the time needed to solve these more complicated MPC problems. Lastly, there is considerable opportunity to improve on the benchmarks we have developed.

6.2 Methodology for Quasi-Static Scheduling of CAL Programs

We have developed a methodology for quasi-static scheduling of dynamic dataflow specifications in the CAL language. Our approach is based on systematic construction of statically schedulable regions, which are formally and uniquely defined in terms of modeling concepts that underlie CAL. Our approach is applied through a novel integration of three complementary dataflow tools — the CAL parser, TDP, and CAL2C — and

demonstrated on an IDCT module from a reconfigurable video decoder application. After detecting statically schedulable regions (SSRs), we can efficiently make use of available SDF techniques and tools to schedule SSRs in terms of their respective sets of SSR actors.

CAL actor programming and SSR detection allow designers and tools to analyze different forms of concurrency, which can significantly improve the efficiency of circuits and systems for video processing. Our experimental results show that integration of SDF-like regions into CAL2C makes the derived multi-core implementations significantly faster. The overall goal of our work on CAL is to provide an automatic design flow from user-friendly design to efficient implementation of video processing systems.

We also proposed an automatic design flow from user-friendly design to efficient implementation of reconfigurable video coding systems. We have developed tools and techniques to favor both designer productivity and implementation efficiency by combining multiple complementary dataflow languages and environments. We use the CAL language as a concrete framework for representing and demonstrating generalized dataflow design techniques. We have developed a new XML format, called DIFML, to communicate between different dataflow tools. Our approach is a novel integration of the three complementary dataflow tools — the CAL parser, TDP, and CAL2C. Our experimental results have demonstrated significant performance improvement on an MPEG Reconfigurable Video Coding (RVC) decoder.

Important directions for further work include the exploration of CAL-based design, analysis and optimization for other types of hardware platforms beyond multi-core platforms; programmer-directed implementation of SSRs for interactive performance tuning; and SSR transformations (e.g., clustering and decomposition transformations) for opti-

mizing thread granularity.

BIBLIOGRAPHY

[1] J. Eker and J. W. Janneck, “CAL language report, language version 1.0 — document edition 1,” Electronics Research Laboratory, University of California at Berkeley, Tech. Rep. UCB/ERL M03/48, December 2003.

[2] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, “CIL: An Infrastructure for C Program Analysis and Transformation,” in *Proceedings of CC 2002*, April 2002, pp. 213–228.

[3] M. Wipliez, G. Roquier, M. Raulet, J. Nezan, and O. Deforges, “Code generation for the MPEG reconfigurable video coding framework: From CAL actions to C functions,” in *Proceedings Multimedia and Expo, IEEE International Conference*, June 2008, pp. 1049–1052.

[4] E. F. Camacho and C. Bordons, *Model predictive control in the process industry*. Springer, 1995.

[5] S. Puthenpurayil, R. Gu, and S. S. Bhattacharyya, “Energy-aware data compression for wireless sensor networks,” in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April 2007, pp. 45–48.

[6] P. Salmela, R. Gu, S. S. Bhattacharyya, and J. Takala, "Efficient parallel memory organization for turbo decoders," in *In Proceedings of the European Signal Processing Conference*, September 2007, pp. 831–835.

[7] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "Optimized software synthesis for synchronous dataflow," in *Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors*, July 1997.

[8] C. Hsu, M. Ko, and S. S. Bhattacharyya, "Software synthesis from the dataflow interchange format," in *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, Dallas, Texas, September 2005, pp. 37–49.

[9] G. Roquier, M. Wipliez, M. Raulet, J. W. Janneck, I. D. Miller, and D. B. Parlour, "Automatic software synthesis of dataflow program: An MPEG-4 simple profile decoder case study," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, October 2008.

[10] R. M. Karp and R. E. Miller, "Properties of a model for parallel computations: Determinacy, termination, queuing," *SIAM Journal of Applied Math*, vol. 14, no. 6, November 1966.

[11] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress*, 1974.

[12] J. B. Dennis, "First version of a data flow procedure language," Laboratory for Computer Science, Massachusetts Institute of Technology, Tech. Rep., May 1975.

[13] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, pp. 773–799, May 1995.

[14] E. A. Lee and D. G. Messerschmitt, "Synchronous dataflow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[15] R. Lublinerman and S. Tripakis, "Translating data flow to synchronous block diagrams," in *Proceedings of the IEEE Workshop on Embedded Systems for Real-Time Multimedia*, 2008.

[16] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, 2nd ed. CRC Press, 2009.

[17] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static dataflow," *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, February 1996.

[18] J. T. Buck and E. A. Lee, "The token flow model," in *Advanced Topics in Dataflow Computing and Multithreading*, L. Bic, G. Gao, and J. Gaudiot, Eds. IEEE Computer Society Press, 1993.

[19] B. Kienhuis and E. F. Deprettere, "Modeling stream-based applications using the SBF model of computation," *Journal of Signal Processing Systems*, vol. 34, no. 3, 2003.

[20] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya, "Functional DIF for rapid prototyping," in *Proceedings of the International Symposium on Rapid System Prototyping*, Monterey, California, June 2008, pp. 17–23.

[21] C. Hsu and S. S. Bhattacharyya, "Porting DSP applications across design tools using the dataflow interchange format," in *Proceedings of the International Workshop on Rapid System Prototyping*, Montreal, Canada, June 2005, pp. 40–46.

[22] R. Gu, J. W. Janneck, S. S. Bhattacharyya, M. Raulet, M. Wipliez, and W. Plishker, "Exploring the concurrency of an MPEG RVC decoder based on dataflow program analysis," *IEEE Transactions on Circuits and Systems for Video Technology*, 2009.

[23] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubühr, A. Deyhle, A. Hadert, and J. Teich, "A SystemC-based design methodology for digital signal processing systems," *EURASIP Journal on Embedded Systems*, vol. 2007, pp. Article ID 47 580, 22 pages, 2007.

[24] W. Sung, M. Oh, C. Im, and S. Ha, "Demonstration of hardware software codesign workflow in PeaCE," in *Proceedings of the International Conference on VLSI and CAD*, October 1997.

[25] R. Gu, S. S. Bhattacharyya, and W. S. Levine, "Dataflow-based implementation of model-predictive control," in *Proceedings of American Control Conference*, June 2009, pp. 2343–2349.

[26] ——, "Improving the performance of active set based model predictive controls by dataflow methods," in *Proceedings of the 48th IEEE Conference on Decision and Control held jointly with the 2009 28th Chinese Control Conference*, December 2009, pp. 339–344.

[27] Y. Wang and S. Boyd, “Fast model predictive control using online optimization,” in *Proceedings of the IFAC World Congress*, July 2008, pp. 6974–6979.

[28] L. G. Bleris, J. Garcia, M. G. Arnold, and M. V. Kothare, “Towards embedded model predictive control for system-on-a-chip applications,” *Journal of Process Control*, vol. 16, no. 3, March 2006.

[29] K. Edlund, L. E. Sokoler, and J. B. Jorgensen, “A primal-dual interior-point linear programming algorithm for MPC,” in *Proceedings of the 48th IEEE Conference on Decision and Control held jointly with the 2009 28th Chinese Control Conference*, December 2009, pp. 351–356.

[30] Y. K. Chen, C. Chakrabarti, S. S. Bhattacharyya, and B. Bougard, “Signal processing on platforms with multiple cores: Part 1–overview and methodologies,” *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 24–25, November 2009.

[31] J. Richalet, A. Rault, J. L. Testud, and J. Papon, “Model predictive heuristic control: application to industrial processes,” *Automatica*, vol. 14, no. 2, pp. 413–428, 1978.

[32] C. R. Cutler and B. Ramaker, “Dynamic matrix control—a computer control algorithm,” in *Proceedings of the Joint Automatic Control Conference*, 1980.

[33] A. Bemporad, M. Morari, V. Dua, and E. Pistikopoulos, “The explicit linear quadratic regulator for constrained systems,” *Automatica*, vol. 38, no. 1, pp. 3–20, January 2002.

[34] H. Chung, E. Polak, and S. Sastry, "An accelerator for packages solving discrete-time optimal control problems," in *Proceedings of the IFAC World Congress*, July 2008, pp. 14 295–14 300.

[35] K. V. Ling, S. P. Yue, and J. M. Maciejowski, "An FPGA implementation of model predictive control," in *Proceedings of the American Control Conference*, June 2006.

[36] L. G. Bleris, P. D. Vouzis, M. G. Arnold, and M. V. Kothare, "A co-processor FPGA platform for the implementation of real-time model predictive control," in *Proceedings of the American Control Conference*, June 2006.

[37] J. J. More and G. Toraldo, "Algorithms for bound constrained quadratic programming problems," *Numer. Math.*, vol. 55, no. 4, pp. 377–400, 1989.

[38] P. A. Absil and A. L. Tits, "Newton-KKT interior-point methods for indefinite quadratic programming," *Computational Optimization and Applications*, vol. 36, no. 1, pp. 5–41, January 2007.

[39] R. Milman and E. J. Davison, "A fast MPC algorithm using nonfeasible active set methods," *Journal of Optimization Theory and Applications*, vol. 139, no. 3, pp. 591–616, 2008.

[40] P. E. Gill, W. Murray, and M. Wright, *Practical Optimization*. Academic Press, London, UK, 1981.

[41] J. M. Maciejowski, *Predictive Control with Constraints*. Prentice Hall, 2002.

[42] A. Zheng, "Reducing on-line computational demands in model predictive control by approximating qp constraints," *Journal of Process Control*, vol. 9, pp. 279–290, August 1999.

[43] R. Gu, J. Janneck, M. Raulet, and S. S. Bhattacharyya, "Exploiting statically schedulable regions in dataflow programs," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Taipei, Taiwan, April 2009, pp. 565–568.

[44] S. S. Bhattacharyya, R. Leupers, and P. Marwedel, "Software synthesis and code generation for DSP," *IEEE Transactions on Circuits and Systems — II: Analog and Digital Signal Processing*, vol. 47, no. 9, pp. 849–875, September 2000.

[45] M. Wipliez, G. Roquier, M. Raulet, J.-F. Nezan, and O. Deforges, "Code generation for the MPEG reconfigurable video coding framework: From CAL actions to C functions," in *Proceedings of the IEEE International Conference on Multimedia and Expo*, 2008.

[46] M. Wipliez, G. Roquier, and J. Nezan, "Software code generation for the RVC-CAL language," *Journal of Signal Processing Systems*, June 2009. [Online]. Available: http://dx.doi.org/10.1007/s11265-009-0390-z

[47] "Open RVC CAL compiler." [Online]. Available: http://sourceforge.net/apps/trac/orcc/

[48] S. S. Bhattacharyya, J. Eker, J. W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet, "Overview of the MPEG reconfigurable video coding framework,"

Journal of Signal Processing Systems, June 2009. [Online]. Available: http://dx.doi.org/10.1007/s11265-009-0399-3

[49] C. Lucarz, M. Mattavelli, J. Thomas-Kerr, and J. Janneck, "Reconfigurable media coding: A new specification model for multimedia coders," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, October 2007.

[50] S. S. Bhattacharyya, G. Brebner, J. Eker, J. W. Janneck, M. Mattavelli, C. von Platen, and M. Raulet, "OpenDF — a dataflow toolset for reconfigurable hardware and multicore systems," *ACM SIGARCH Comput. Archit. News*, vol. 36, no. 5, 2008. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00398827/en/

[51] C. v. Platen and J. Eker, "Efficient realization of a CAL video decoder on a mobile terminal," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, October 2008.

[52] J. Boutellier, V. Sadhanala, C. Lucarz, P. Brisk, and M. Mattavelli, "Scheduling of dataflow models within the reconfigurable video coding framework," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, October 2008.

[53] M. Li, H. Wang, and P. Li, "Tasks mapping in multi-core based system: hybrid ACO&GA approach," in *Proceedings of the International Conference on ASIC*, October 2003.

[54] R. Ennals, R. Sharp, and A. Mycroft, "Task partitioning for multi-core network processors," in *Proceedings of the International Conference on Compiler Construction*, April 2005.

[55] T. H. Cormen, C. Stein, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.

[56] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations," *Journal of Design Automation for Embedded Systems*, vol. 2, no. 1, pp. 33–60, January 1997.

[57] W. Plishker, N. Sane, and S. S. Bhattacharyya, "A generalized scheduling approach for dynamic dataflow applications," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Nice, France, April 2009, pp. 111–116.

[58] M. Ko, C. Zissulescu, S. Puthenpurayil, S. S. Bhattacharyya, B. Kienhuis, and E. Deprettere, "Parameterized looped schedules for compact representation of execution sequences in DSP hardware and software implementation," *IEEE Transactions on Signal Processing*, vol. 55, no. 6, pp. 3126–3138, June 2007.

[59] J. Janneck, I. Miller, D. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing hardware from dataflow programs," *Journal of Signal Processing Systems*, June 2009. [Online]. Available: http://dx.doi.org/10.1007/s11265-009-0397-5

[60] *MPEG video technologies – Part 4: Video tool library*, ISO/IEC FDIS 23002-4, 2009.

[61] *MPEG systems technologies – Part 4: Codec Configuration Representation*, ISO/IEC FDIS 23001-4, 2009.

[62] S. Vinoski, "Concurrency with Erlang," *IEEE Internet Computing*, vol. 11, no. 5, pp. 90–93, 2007.

[63] T. Chen and Y. K. Chen, "Challenges and opportunities of obtaining performance from multi-core CPUs and many-core GPUs," in *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, April 2009.

[64] G. Roquier, M. Wipliez, M. Raulet, J. Janneck, I. Miller, and D. Parlour, "Automatic software synthesis of dataflow program: An MPEG-4 simple profile decoder case study," in *Proceedings of IEEE Workshop on Signal Processing Systems*, October 2008, pp. 281–286.

[65] C. Hsu, I. Corretjer, M. Ko., W. Plishker, and S. S. Bhattacharyya, "Dataflow interchange format: Language reference for DIF language version 1.0, user s guide for DIF package version 1.0," Institute for Advanced Computer Studies, University of Maryland at College Park, Tech. Rep. UMIACS-TR-2007-32, June 2007.

[66] E. M. Gagnon and L. J. Hendren, "SableCC, an object-oriented compiler framework," in *Proceedings of TOOLS (26)*, 1998, pp. 140–154.

[67] B. Kienhuis and E. F. Deprettere, "Modeling stream-based applications using the SBF model of computation," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, September 2001, pp. 385–394.

[68] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, "Optimizing synchronization in multiprocessor DSP systems," *IEEE Transactions on Signal Processing*, vol. 45, no. 6, pp. 1605–1618, June 1997.

[69] J. T. Buck, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," Ph.D. dissertation, Electrical Engineering and Computer Sciences Department, University of California, Berkeley, 1993. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1993/2429.html

[70] J. Piat, M. Raulet, M. Pelcat, P. Mu, and O. Déforges, "An extensible framework for fast prototyping of multiprocessor dataflow applications," in *IDT'08: Proceedings of the 3rd International Design and Test Workshop*, Monastir, Tunisia, December 2008.

[71] R. Gu, J. Janneck, M. Raulet, and S. S. Bhattacharyya, "Exploiting statically schedulable regions in dataflow programs," *Journal of Signal Processing Systems*, January 2010. [Online]. Available: http://www.springerlink.com/content/7828n20m31186635/

[72] Z. Navabi, *Verilog Digital System Design: Register Transfer Level Synthesis, Testbench, and Verification*. McGraw Hill, 2006.

[73] S. Sutherland, S. Davidmann, and P. Flake, *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modelingn*, 2nd ed. Springer, 2006.

[74] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Proceedings of the International Workshop on Rapid System Prototyping*, San Jose, California, USA, 2006, pp. 151–162.

[75] B. Bhattacharya and S. S. Bhattacharyya, “Parameterized dataflow modeling for DSP systems,” *IEEE Transactions on Signal Processing*, vol. 49, no. 10, pp. 2408–2421, October 2001.

[76] P. K. Murthy and E. A. Lee, “Multidimensional synchronous dataflow,” *IEEE Transactions on Signal Processing*, vol. 50, no. 8, pp. 2064–2079, August 2002.