Final Report for Austrian Marshall Plan Foundation

TITLE: DATAFLOW BASED MODELING AND OPTIMIZATION FOR VARIANT EMBEDDED APPLICATIONS

NAME: Ruirui Gu

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

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 used in the final research report include real-time system applications of reconfigurable video coding (RVC) [2] and model predictive control (MPC) [3].

Previously, because embedded systems were highly limited in computation capability, memory size, and power consumption constraints, much research was dedicated to making the best use of limited system resources. Examples include techniques for energy efficient system design [4] and memory size efficiency [5]. 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.

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 multicore 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.

In general, the process of developing an embedded system is divided into two phases: design and implementation. 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 the 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.

Dataflow modeling techniques underlie many popular graphical tools for digital signal processing (DSP) system design (e.g., see [6]). 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) [7], 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.

The rest of report is organized as follows: Chapter 2 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 [8], DIF [7] and CAL2C [9]. Chapter 3 presents an in-depth case study on 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 4 describes a general framework called reactive, control-integrated dataflow modeling for analyzing and improving algorithms used for MPC and their hardware implemen-

tations. Our work describes modeling and analysis tools to facilitate implementing the 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 a class of MPC problems. A summary of current progress and future work to explore parallelism are described in Chapter 5.

Our research work explores systematic exploitation of parallelism in embedded computing systems. From system design to hardware implementation, parallelism can be explored in a hierarchical way. By operating at a high level of abstraction and taking into account different levels and forms of parallelism in a unified way, our methods have the potential for significant performance impact compared to conventional methods, which are often restricted to specialized modes of parallel processing. Simulation and experimental results show that our method can deliver significant improvements in system performance.

Chapter 2

Exploiting Statically Schedulable Regions

2.1 Overview

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 [10]. 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 [11]).

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 and implementation of embedded DSP systems. For example, CAL [8] 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 [12]. 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 sematics 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 [7] 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 [7]. CAL2C [13, 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 [14] and can be downloaded from [15].

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 adaptibility compared to fully static scheduling.

More specifically, 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 improvments in system performance compared to conventional actor-based mapping

approaches.

This chapter is organized as follows. Section 2.2 introduces previous work related to dataflow models, the CAL language, and related efforts on extracting SDF-like parts from dynamic dataflow models. Section 2.3 outlines our methods and notations for translation and analysis across different modeling languages. In Section 2.4, we introduce the concept of SSRs, and develop a detailed procedure for deriving SSRs from CAL networks. Section 2.5 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 2.6 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.

2.2 Related Work

2.2.1 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 [16], Kahn process networks [17], dataflow architectures [18], and dataflow process networks [19]. Synchronous dataflow (SDF) is a specialized form of dataflow that is streamlined for efficient representation of DSP systems [10].

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 [20]). Model 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 [21]) .

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 2.1. Here, A and B represent two actors,

and the numbers shown above the edges represent the rates at which actors produce and consume tokens. For example, A produces two tokens every time it executes and B consumes three tokens 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 2.1 conforms to the SDF model.

A limitation of SDF and related models, such as cyclo-static dataflow [22] and homogeneous SDF (HSDF) [10], 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 [23], stream-based functions [12], enable-invoke dataflow (EIDF) [24], and the CAL actor language [8].

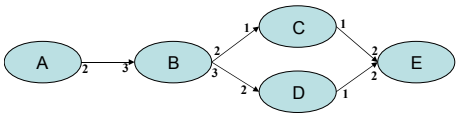


Figure 2.1: A simple example of a dataflow (SDF) model.

2.2.2 DIF

The dataflow interchange format (DIF) is proposed as a standard approach for specifying and integrating arbitrary dataflow-based semantics for DSP system design [25]. The DIF package (TDP) [7, 24] 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 [25].

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 datalfow 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 [7, 24].

For example, TDP includes a transformation that converts SDF representations into equivalent homogeneous SDF (HSDF) representations based on the algorithm introduced in [10]. Such a transformation can in general expose additional concurrency [26] that is not represented explicitly in the orignal SDF graph. In this chapter, we make use of both generic-graph-based (via Mocgraph) and model-based (via TDP) analysis methods to automatically derive and exploit SSRs from within CAL networks. As we will demonstrate later in this chapter, such extraction and exploitation of SSRs provides a powerful new methodology for optimized implementation of datalfow graphs. In comparison, [26] presents in-depth dataflow based analysis and exploitation of parallelism in the design and implementation of an MPEG RVC decoder, while this chapter focuses on detailed description of the SSR detection algorithm.

Compared to other design tools for representation and transformation of dataflow graphs — such as SysteMoC [27], PeaCE [28], and stream-based functions [12] — 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.

2.2.3 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) [29], uses the CAL actor language [8] for specifying functional components, and dataflow as the composition formalism [30].

An integrated set of tools related to CAL are presented in OpenDF [31]. 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 [32] sketches a method to classify CAL actors into different dataflow classes for efficient scheduling. Boutellier et al. [33] 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 [21]). For example, Li et al. [34] 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. [35] 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 capabile 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.

2.3 Analysis Framework

Our method to optimize implementation of DSP applications combines the advantages of three complementary tools, as shown in Figure 3.4. 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.

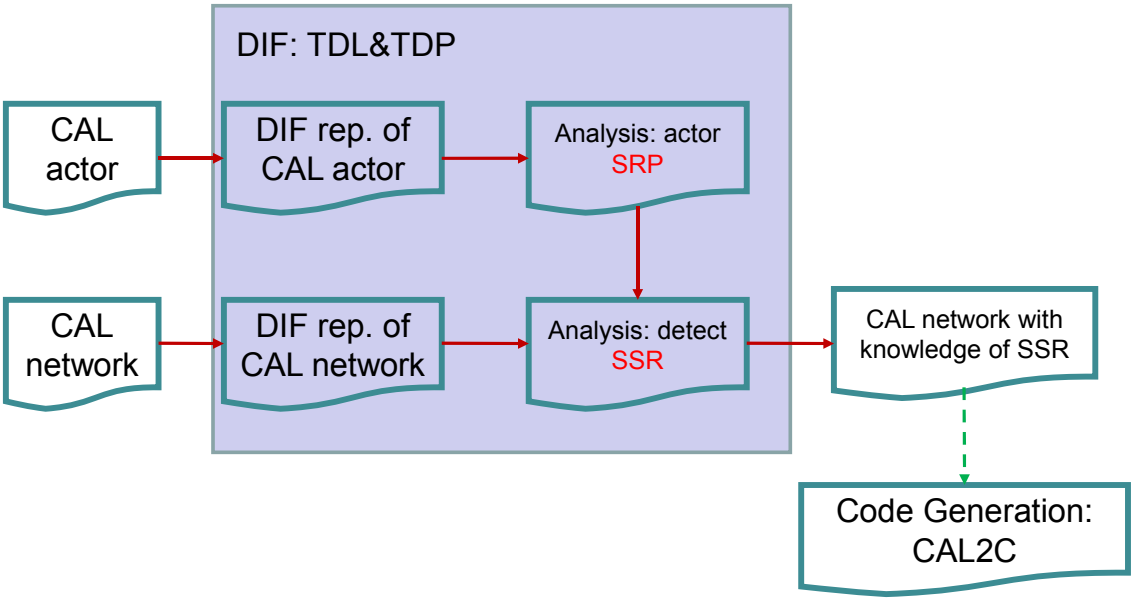


Figure 2.2: Outline of method for optimizing dataflow graph implementation.

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 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 variable 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 actors, 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.

2.4 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 bewteen 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 2.3 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 $\{B.c, B.y\}$.

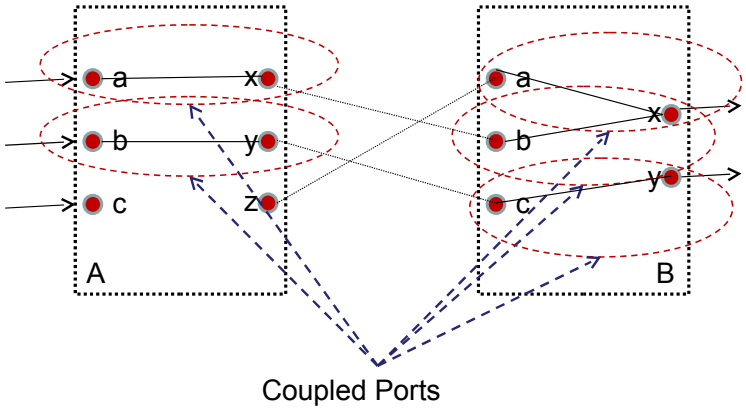


Figure 2.3: An illustration of coupled ports and CRGs.

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 [36]). Intuitively, in an 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 2.4 shows an example of a directed graph, the undirected version of that

graph, the associated weakly connected components.

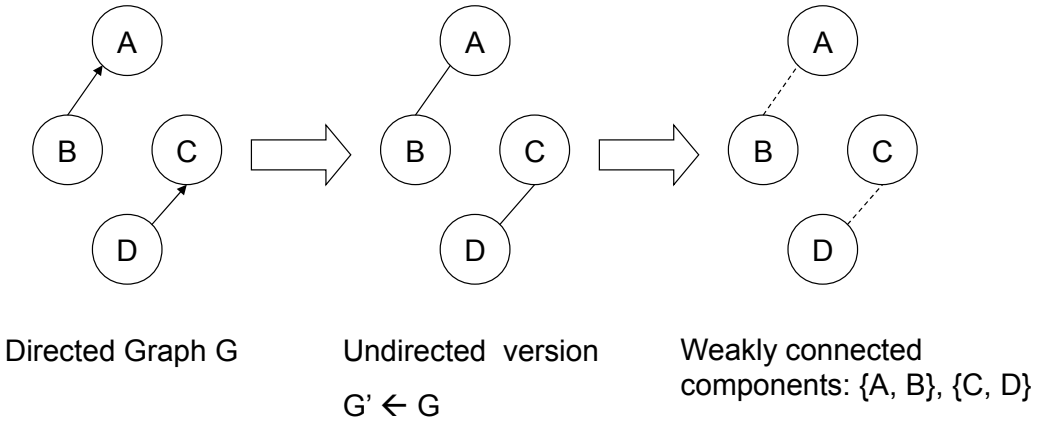


Figure 2.4: An illustration of weakly connected components.

Figure 2.5 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 2.3.

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 $Z = \{p_1, p_2, \ldots, p_n\}$ 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.

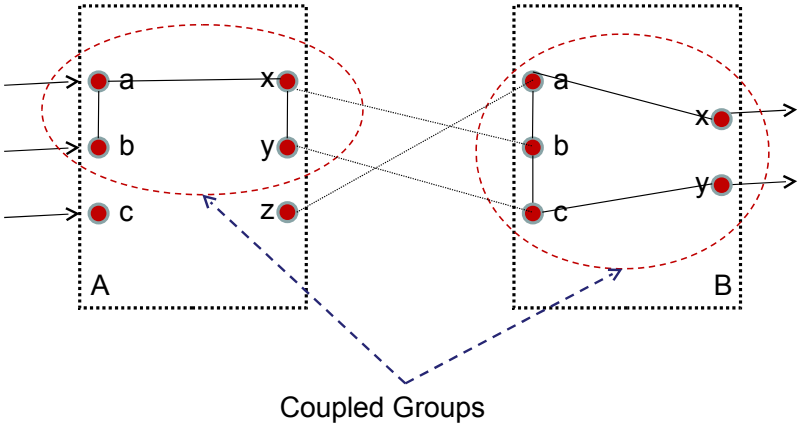


Figure 2.5: An illustration of coupled groups.

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.

SRGs (statically-related groups) can be derived by constructing and analyzing an intermediate graph representation that we call the *static relationship graph*. Given a cou-

pled 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 2.6 shows an example of a static relationship graph and the statically-related groups derived from Figure 2.5.

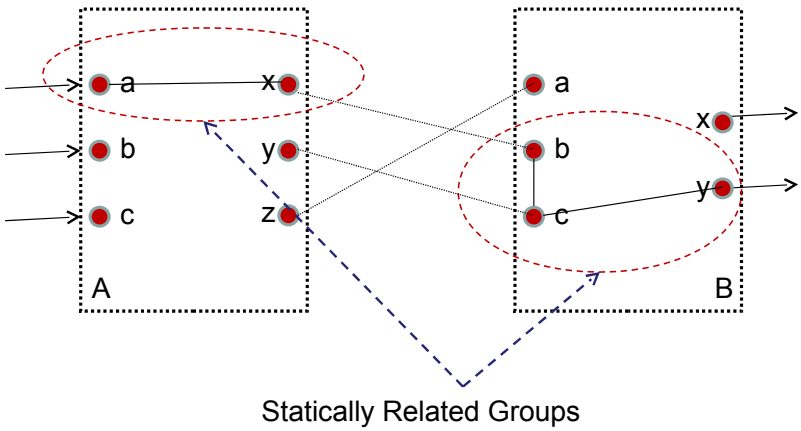


Figure 2.6: An illustration of statically-related groups.

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

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 an 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

$$X_i = \{x_{i,1}, x_{i,2}, \ldots, x_{i,m_i}\}, \tag{2.1}$$

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

$$r_i = \bigcup_{j=1}^{m_i} x_{i,j}. \tag{2.2}$$

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

Figure 2.7 shows an example of an SRG graph and the obtained statically schedulable region.

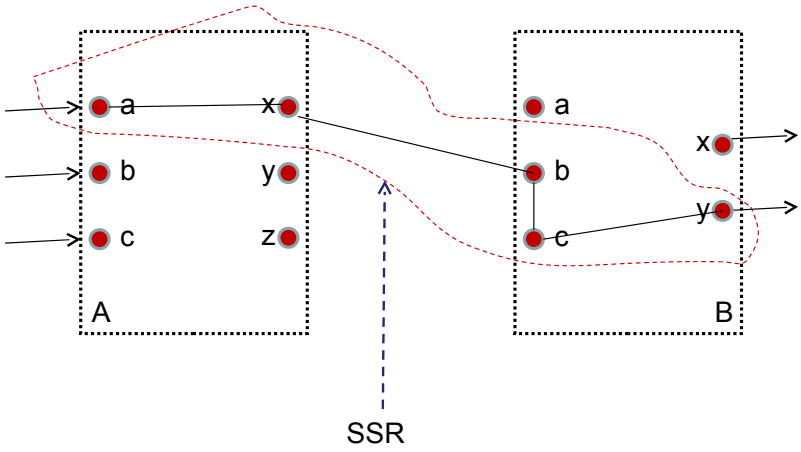


Figure 2.7: An illustration of a statically schedulable region.

2.5 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 propsoed design flow.

In order to apply SDF scheduling techniques to an SSR, we first construct a set of one or more SDF 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.

2.5.1 IDCT Example

Figure 3.6 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.

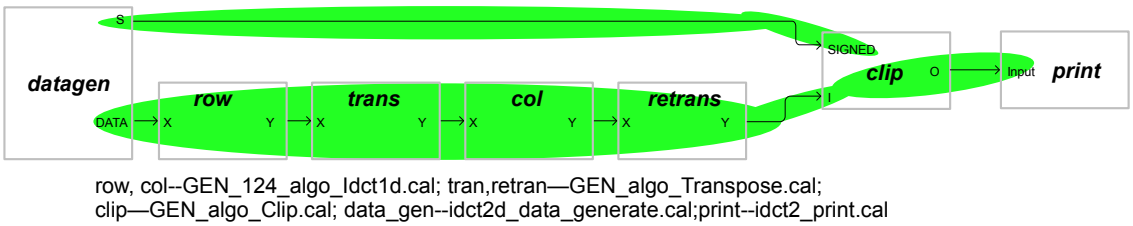


Figure 2.8: SSRs in the IDCT subsystem.

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 straightforward mapping for this case is shown in Figure 3.6. 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 [37] [38], 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 2.9. 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 [39].

In the schedule tree shown in Figure 2.9, 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 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.

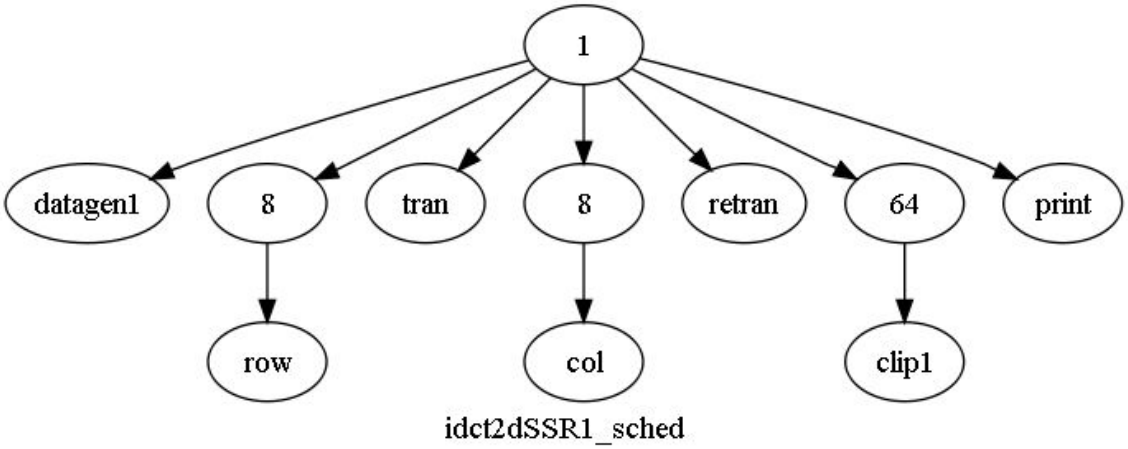


Figure 2.9: Schedule tree for an SSR in the IDCT example.

2.6 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.

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 3.1. 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.

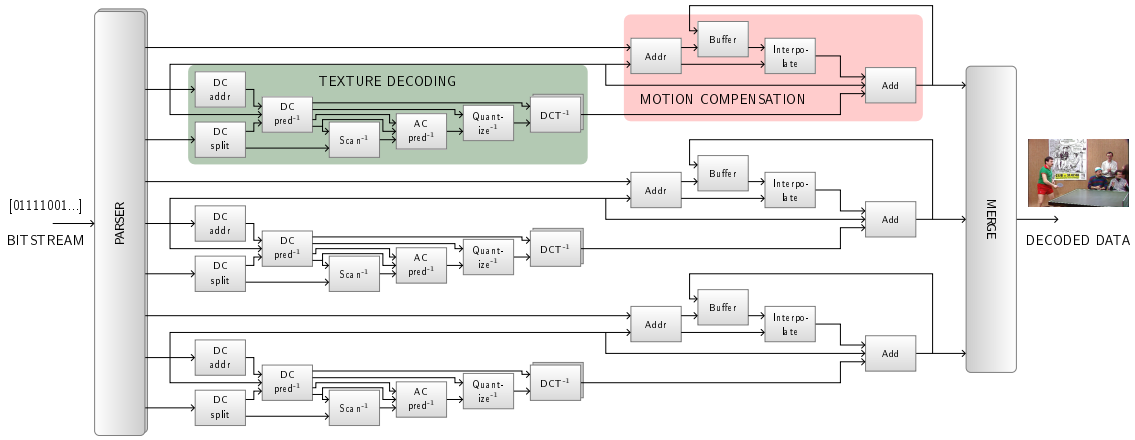


Figure 2.10: A block diagram of an MPEG RVC decoder.

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 presented 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 5 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 CIF sequences with size 624x352. As shown in Table 2.1, 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.

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

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

Table 2.1: MPEG-4 SP decoder performance for 624x352 sequence.

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 3

Exploring the Concurrency of an MPEG RVC Decoder

3.1 Overview

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, 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 [29]. 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 [7]). 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 [29], [14], [40] cover related aspects of reconfigurable video coding and CAL-oriented tools. In particular, [29] gives an overview of the overall RVC framework; Reference [14] provides details on the software code generator CAL2C; and Reference [40] 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 [29], [14] and [40].

Using CAL as a concrete design representation framework, this chapter places em-

phasis 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 representations 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 3.2 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 3.3 analyzes inter-actor concurrency obtained from CAL specifications from the viewpoint of both hardware and software implementation. Section 3.4 proposes techniques for cross actor-optimization that enhance multi-core system performance. Simulation results are also presented in this section.

3.2 Background

3.2.1 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 [29]. 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 [41] and MPEG-C part 4 [42] 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.

3.2.2 Dataflow language

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 [16], Kahn process networks [17], and dataflow architectures [18]. Synchronous dataflow (SDF) is a specialized form of dataflow that is streamlined for efficient representation of DSP systems [10]. 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. 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.

In DSP-oriented dataflow graphs, vertices (*actors*) represent computations of arbitrary complexity, and each 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 that 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).

A limitation of SDF and related models, such as cyclo-static [22] and single-rate [43] dataflow, 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 stream-based functions [44], functional DIF [24], and the CAL actor language [8] that is targeted in this chapter.

3.2.3 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 [45]. 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.

3.2.4 The CAL language

CAL is a dataflow- and actor-oriented language that describes algorithms by using a set of encapsulated functional components (actors) or functional units (FUs) in RVC that communicate with one another based on dataflow semantics. In CAL, an actor is a modular component that encapsulates its own state. The state of an actor cannot be shared with other actors. Thus, an actor cannot modify the state of another actor.

The behavior of a CAL actor is defined in terms of a set of *actions*. The operations an action can perform are consuming (reading) input tokens, modifying internal state, and producing output tokens. The topology of the connections between input and output ports of actors constitute what is called a network of actors. Compared to actors, which can be of arbitrary functional complexity, edges — connections between actors — are conceptually simpler. The only interaction an actor has with other actors is through input and output ports that connect to dataflow graph edges.

CAL actors are specified in terms of *actions*. Each action of an actor defines the kind of transitions that internal states can undergo. An action can only be executed (*fired*) under specific conditions; these conditions can be specified in terms of (1) the availability of input tokens, (2) the values of input tokens, (3) the state of the enclosing actor or (4) the priority of the action. In an actor, actions are executed sequentially — that is, only

one action is executed at a time for a given actor.

RVC uses the CAL actor language [8] as the language for specifying FUs, and the FU network language for the dataflow composition [42]. CAL is supported by a portable interpreter infrastructure called OpenDF that can simulate a hierarchical network of actors. Some tools related to CAL can be found in OpenDF [31]. Among them, we are especially interested in the code generators that translate CAL into C or hardware description language (HDL) code. In addition to the strong encapsulation afforded by the actor description, the dataflow model also makes much more algorithmic parallelism explicit. This provides the unique opportunity to apply the wide range of techniques used to implement dataflow systems to the realization of video coding algorithms on a variety of platforms. In particular, platforms will differ in their degrees of parallelism, which gives rise to the challenging problem of matching the concurrency of the decoder specification with the parallelism of the computing machine that is executing it.

3.2.5 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 [46]). 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 achievable 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.

3.3 Inter-actor concurrency analysis

3.3.1 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 3.1. In Figure 3.1, 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 3.1, 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.

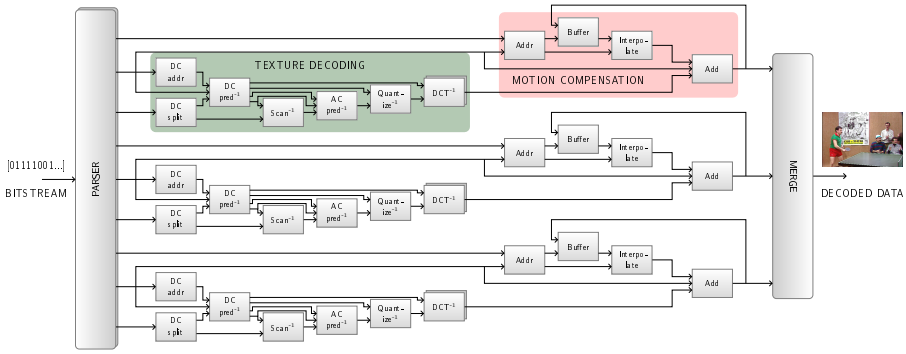


Figure 3.1: An RVC block diagram of an MPEG-4 Simple Profile decoder.

3.3.2 Data parallelism inside CAL networks

In Figure 3.1, 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.

3.3.3 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 [21]). 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.

3.3.4 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 [40]. 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 [40] 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 [2, 47] 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.

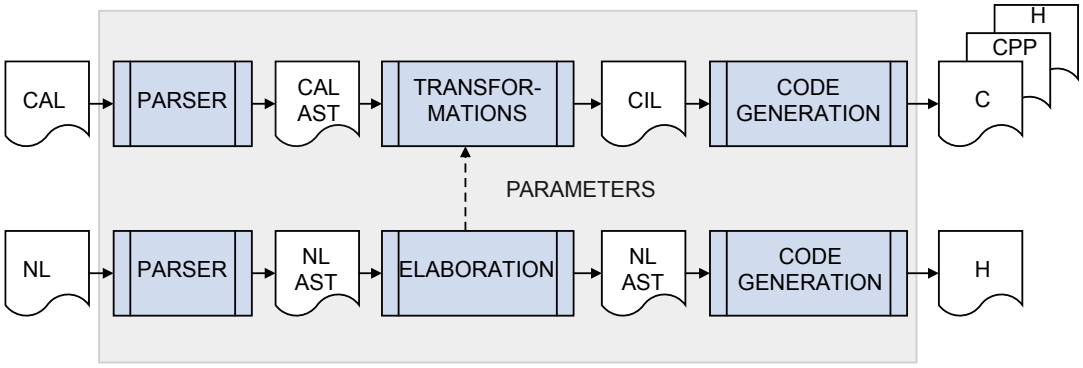


Figure 3.2: CAL2C compilation process: The action translation process starts with an abstract syntax tree (AST) derived from the CAL source code; the transformed CAL AST is expressed in the C intermediate language (CIL) [1], where CAL functional constructs are replaced by imperative ones.

In CAL2C, software synthesis from a CAL network includes two parts: actor transformation [2] and network transformation [47]. 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 [2]. 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 3.2 illustrates how CAL2C works. For further details on CAL2C, we refer the reader to [47].

In [47], 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 3.3.

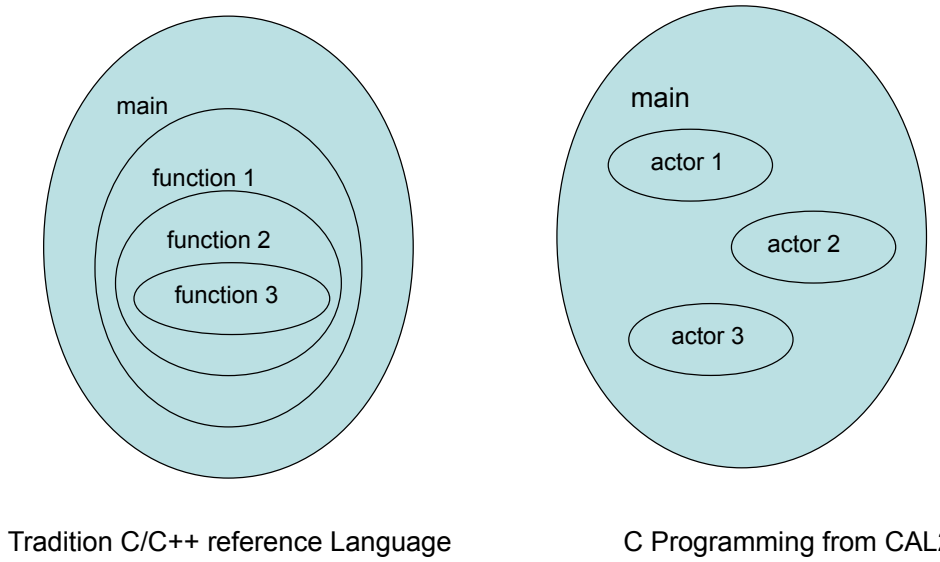


Figure 3.3: Comparison between direct-C/C++-based implementation and implementation using CAL2C.

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.

3.4 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.

3.4.1 DIF and network analysis capability

In this section, we present our application of the dataflow interchange format (DIF) package [7, 43], 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 [48]. For the complete DIF language grammar and a detailed syntax description, we refer the reader to [43].

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 [10]. 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 3.4.3, we develop the concept of SSRs in details.

Compared to other design tools for representation and transformation of dataflow graphs — such as SysteMoC [27], PeaCE [28], and stream-based functions [44] — 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.

3.4.2 Interface between DIF and CAL

Our method to optimize implementation of DSP applications combines the advantages of three complementary tools, as shown in Figure 3.4. 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.

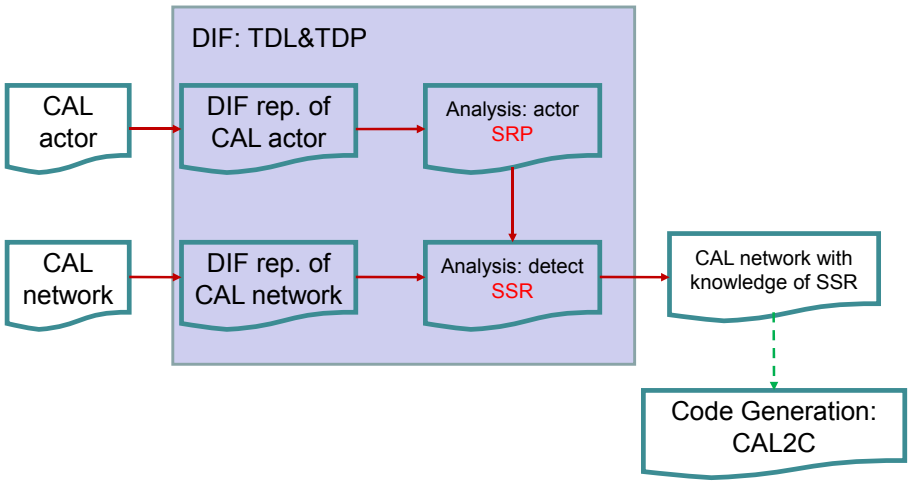


Figure 3.4: Overview of our CAL- and DIF-based method for optimizing dataflow graph implementation. SRP represents statically related port and SSR represents statically related region.

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.

3.4.3 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) [10] 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 3.5, 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 [26].

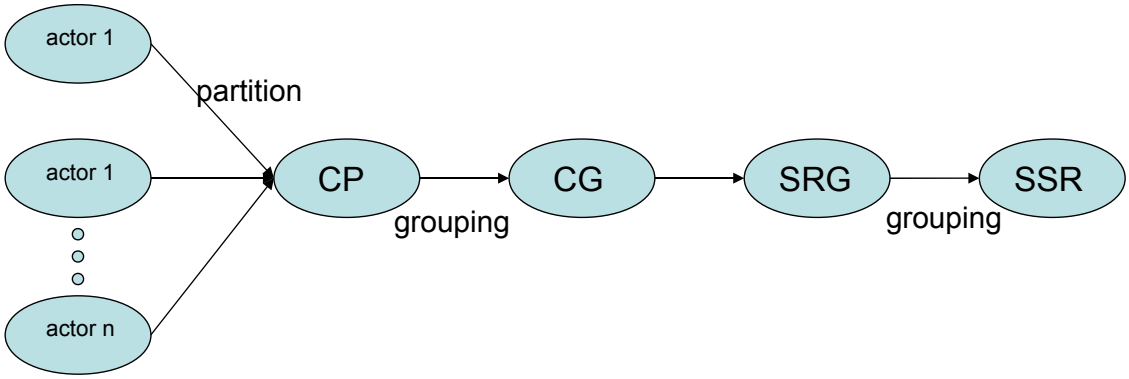


Figure 3.5: SSR detection in PCG.

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.

3.4.4 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 3.6 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

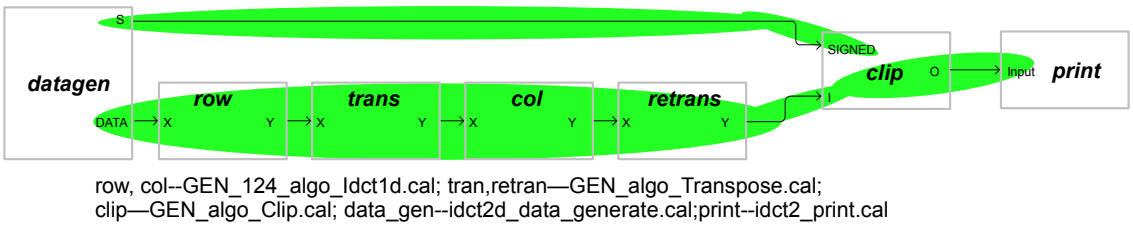


Figure 3.6: SSRs in the IDCT subsystem.

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.

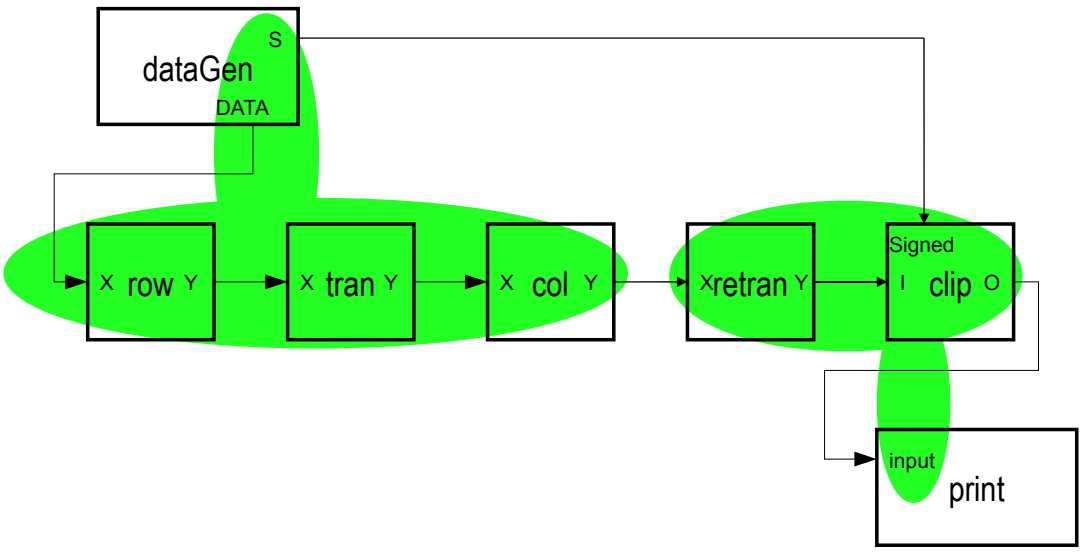
DMA is helpful for intra-chip data transfer in our implementation on multi-core processors, where each processing element is equipped with a local memory and DMA

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 3.7. 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 3.7, 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 3.6. 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 [49] [38]. An example of scheduling of SSRs, including the actor *clip*, is shown in Chapter 2.



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 3.7: 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

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 3.7.

The second version (V2) uses CAL2C-SSR. This version exploits the SSRs illustrated in Figure 3.6, 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 3.8, 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.

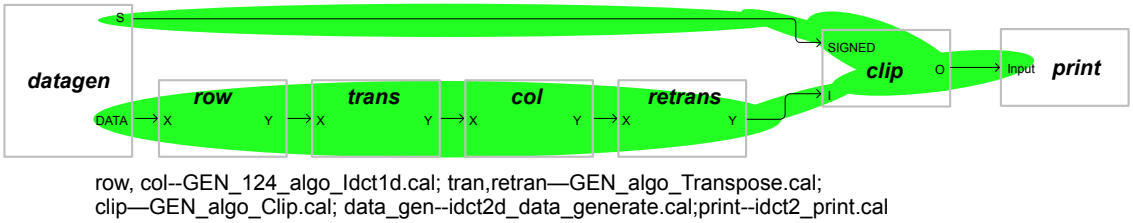


Figure 3.8: IDCT subsystem with one SSR.

We experimented with all three *IDCT* versions using Microsoft Visual Studio. The results are shown in Figure 3.9. 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.

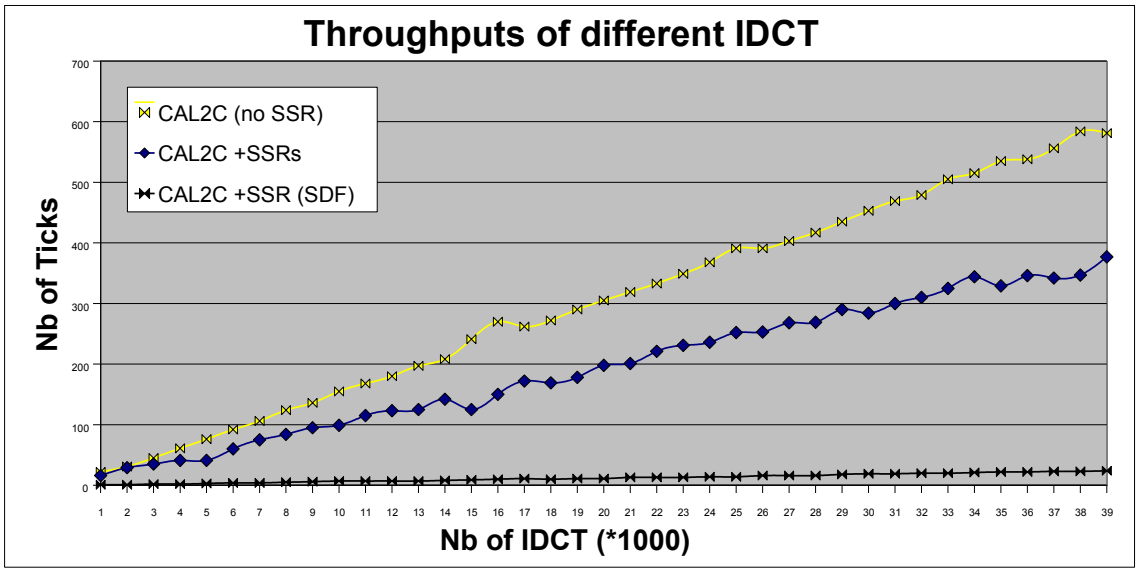


Figure 3.9: Results: clock cycles vs number of iterations.

3.4.5 Concurrency analysis of the MPEG-4 SP decoder

When we analyze the MPEG-4 SP decoder in Figure 3.1 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 3.1, 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 [14]. 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 [50] 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

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

Table 3.1, 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 [51]. Further exploration on this class of "granularity-adjustment" transformations for SSRs is a useful direction for

further work.

Chapter 4

Methods for Efficient Implementation of Model Predictive Control

4.1 Overview

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 [52]. As a result, there has been considerable research aimed at speeding up the computation of optimal controls. Most of this research has concentrated on improving the algorithms. Relatively little work [53] has been devoted to improving the implementation of the algorithms. But the two go hand in hand. For example, Edlund et al. [54] have reduced the time to complete the computations in a specific MPC application by a factor of more than 10 by carefully optimizing the implementation 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, multicore 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 tradeoffs 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 [55]. 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 aims to provide a dataflow-based framework to model and analyze computationally intensive control applications and to improve their performance by taking advantage of rapidly developing parallel distributed systems.

In earlier work [56] and [57] we described the basic framework and applied it to develop 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.

This chapter reports two further developments. The first is straightforward. We have improved the benchmarks for testing our implementations. This is important because bet-

ter benchmarks result in more accurate estimates of the time needed for the computations. The second improvement is in two parts. 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.

The following section of this chapter briefly surveys related work. This is followed by a description of how to apply dataflow methods of modeling and analysis to MPC problems. The next section describes in detail the application of these techniques to reduce the time required to do the Newton-KKT part of the MPC calculations–identified as the bottleneck in the computations for the class of MPC problems under consideration– by means of parallel computation. An important component of this is an exploration of ways in which multicore processors can be used to reduce the time required by Gaussian elimination. The last section contains conclusions and suggestions for further research.

4.2 RELATED WORK

4.2.1 Control 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 [58][59]. 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 appli-

cation 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 [3].

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 [52]. One approach to addressing this problem has been to compute the control law off-line and store it as a lookup table [60]. 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 [52, 61].

A few researchers have addressed the implementation of MPC. Ling et al. [62] demonstrated that a "reasonably sized constrained MPC Controller" could be implemented on a modest FPGA chip. Bleris et al. [63] have proposed a computing architecture that is specifically designed for MPC. Furthermore, they have proposed a design framework for application specific processor implementation [53]. 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.

4.2.2 Embedded Signal Processing Background

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 [16], Kahn process networks [17], and dataflow architectures [18]. Synchronous dataflow (SDF) is a specialized form of dataflow that is streamlined for efficient representation of DSP systems [10]. 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. 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.

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* ac-

tors (actors that have no input edges).

A limitation of SDF and related models, such as cyclo-static [22] dataflow, 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 stream-based functions [44], functional DIF [24], and the CAL actor language [8]. In this chapter, we describe a new dynamic dataflow modeling technique, called reactive, control-integrated dataflow (RCDF), that appears particularly promising for MPC applications. Our approach is more specialized compared to other dynamic dataflow techniques, but for MPC, this specialization can be exploited in useful ways to streamline the implementation process.

Note that in addition to their formal properties, DSP-oriented dataflow models provide different kinds of software architectures for working with signal processing computations (of which control system implementations form an important sub-class). This kind of representation can help to structure subsequent phases of design, simulation, verification, testing, and implementation *regardless of whether the underlying model of computation is explicitly supported by an off-the-shelf design tool*. This is true especially in the area of embedded systems, including embedded control, where designers are often will-

ing to explore specialized, application/architecture-driven analysis techniques that may provide streamlined performance, power consumption, cost, or robustness.

4.3 DATAFLOW BASED FRAMEWORK FOR MODEL PREDICTIVE CONTROL

The dataflow framework provides a complete solution from system modeling to optimized implementation, as shown in Figure 4.1. First of all, the control algorithm is modeled 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 [7].

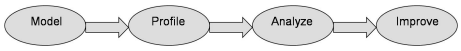


Figure 4.1: Dataflow Framework for efficient system implementation

To facilitate efficient implementation of MPC applications, we have introduced 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 [56]. 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 modeling different reactive control structures such as switch and reset.

The general structure of MPC is shown in Figure 4.2. all the MPC algorithms possess common elements and different options can be chosen for each element giving rise to different algorithms.

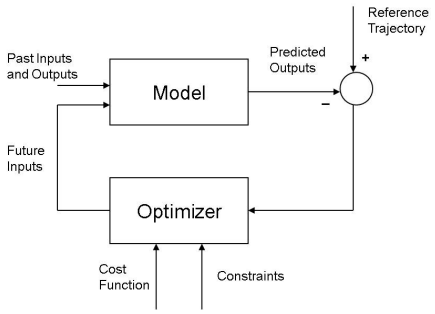


Figure 4.2: Basic Structure of Model Predictive Control

In practice, many MPC problems involve repeated solutions of:

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

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 a $n\times 1$ 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 benchnark 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 problem $< P >$:

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

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 $Nm \times 1$.

In previous work [56, 57], we modeled, analyzed, and improved the implementation of both the Newton KKT and active set methods for solving the general QP problem. The details can be found in [56]. As a first test we used the optimized versions of the Newton-KKT and active set methods to our new benchmark problems. The results are shown in Table 1.

The simulation results were obtained using a desktop comuter with a 1.30GHz pro-

Table 4.1: Simulation results for MPC problems: execution time for different scenarios (sec)

statistics	mean	variance
seq	0.52656	0.053714
$newton - KKT_p$	0.16953	0.0041189
$active - set_p$	0.18098	0.0012329

cessor. seq denotes sequential implementation of the Newton KKT algorithm; $Newton - KKT_p$ is our improved implementation of the Newton KKT algorithm; and $active - set_P$ is our improved implementation of the active set method.

4.4 Newton KKT Incorporating a Parallel Linear System Solver

4.4.1 Newton KKT

Figure 4.3 illustrates a model, developed in [56], of the Newton KKT algorithm based on RCDF. We implemented communication between actors based on the dataflow model, however implementation of each actor used purely sequential programming. As shown in Figure 4.3, there are seven actors in the system, and each actor is responsible for a certain function. The function of each actor is described in brief as follows:

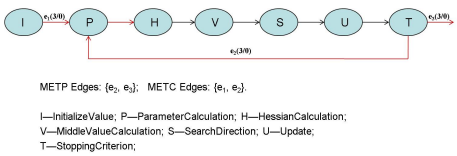


Figure 4.3: RCDF model of Newton KKT algorithm

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

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

H—The actor H is used to compute the modified Hessian matrix. It functions only in the condition that the Hessian matrix has some eigenvalues equal to 0.

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. This is where Newton's method is used.

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.

Since the actors are divided based on functionality, code size is different from one actor to another. The simplest actor may be composed of only one addition. A much more complex actor may be expanded as a dataflow subgraph, such as that represented by the hierarchical actor U in Figure 4.3.

We conduct simulations in MATLAB to evaluate the time each actor consumes. From the profiling result in Table 4.2, it is obvious that H, S and U are computation intensive actors compared with actor V.

In our previous work, we applied functional parallelism to the actors H and S. The modified RCDF model is shown in Figure 4.4.

We carefully transformed actor U to derive an efficient implementation for it, as shown in Figure 4.5. Since the original dataflow model was based on a sequential pro-

Table 4.2: execution time in seconds for different actors in Newton KKT

$ncond$	0	3	6	9	12
H	0.015625	0.031250	0.015625	0.015625	0.031250
S	0.015625	0.042834	0.042834	0.031256	0.018420
U	0.015625	0.015625	0.011230	0.015625	0.023680
V	0.000000	0.000000	0.000000	0.000000	0.000000

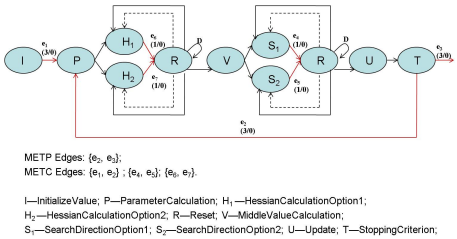


Figure 4.4: Modified RCDF model of Newton KKT algorithm: modified actor H and actor S

gramming language, this is a sequential model in terms of computation.

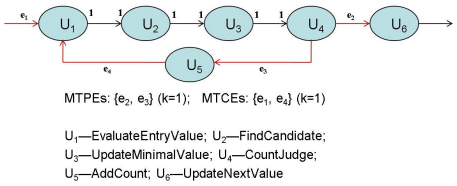


Figure 4.5: Sequential RCDF model of actor U

We transformed the dataflow model in order to make use of parallelism. The transformed dataflow model is shown as Figure 4.6. In the transformed RCDF model, actors U_{t1} and actor U_{t2} contain independent computations for a set of data. If these computations are implemented in a multi-processor system, we can improve system performance.

4.4.2 Parallel Linear System Solver

From Table 4.2, 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

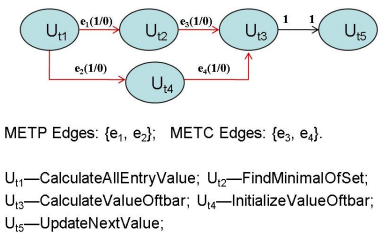


Figure 4.6: Modified RCDF model of actor U

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 importance 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.

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 multicore 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 we wish to solve has the form

$$Ax = b \tag{4.1}$$

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 4.7. In Figure 4.7, 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 4.7: Sequential Program of GE.

Grid computation, such as in ScaLAPACK, is typical for parallel Gaussian Elimination. In this way, key computations are identified and then distributed in a processor matrix.

Figure 4.8 presents our RCDF model of one processing unit for Gaussian Elimination.

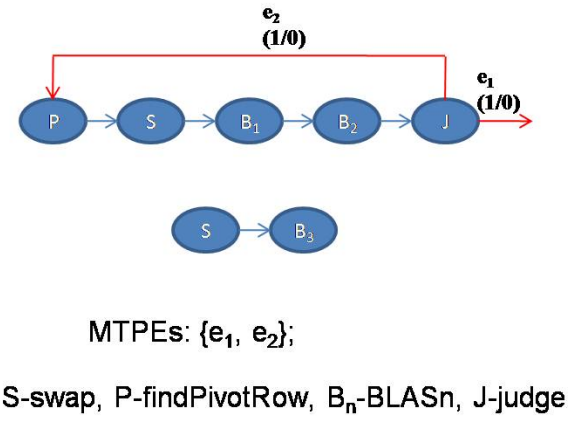


Figure 4.8: RCDF model of Gaussian Elimination on Single Processing Unit

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 imlementation 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 commu-

nicate 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 4.9. In this model, the processor matrix for GE is 2x2. 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.

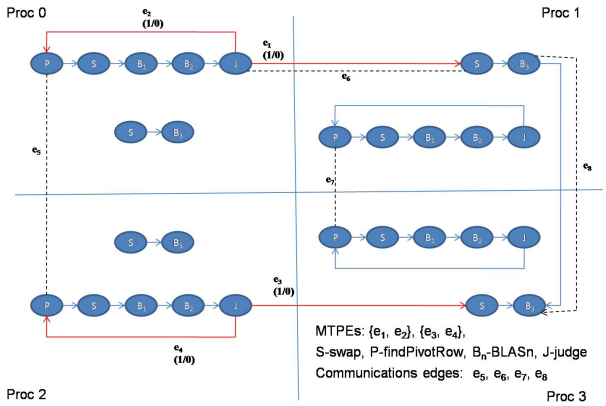


Figure 4.9: RCDF model of Gaussian Elimination on Four Processing Units

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 4.10.

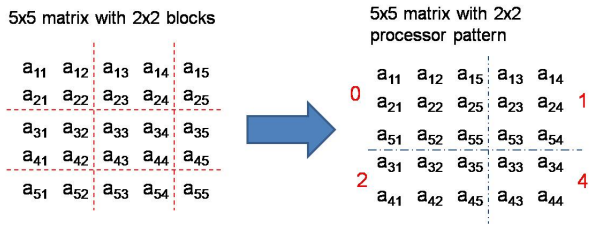


Figure 4.10: 2-D Block cyclic distribution of computations onto parallel processors.

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 are communication between two processors, as indicated by the communication edges shown in Figure 4.9, the communication overhead estimate is added onto the total execution time.

By applying the Schur complement to problem $< P >$ [64], we decreased the dimensions of the linear system from 1200x1200 to 400x400. 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. The simulation results are shown in Table 4.3. The numbers in the table were determined in the following way. The benchmark QPs set up earlier involving $\hat{A}$, $\hat{C}$, $\hat{d}$ with $\hat{u}$ as the unknown to be computed were input to our improved implementation of

Table 4.3: 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

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 in the various ways indicated in Table III to obtain the indicated results.

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 time increases with the increasing number of processors.

In our simulation, we assume that all the processors are homogenous, 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.

Table 4.4: 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

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 III. The simulation results are shown in Table 4.4.

Simulation results indicate that the system performance achieves its peak when the block size is 40. This is also 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 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 4.5. In Table 4.5, 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.

The performance of Newton-KKT with parallel Gaussian elimination will be further improved if we use more processors than 16. However, with 2×8 processors, the actor S

Table 4.5: Simulation results for MPC problems with parallel Gaussian Elimination.

statistics	mean	variance
seq	10.3689	1.309000
newton-KKT	6.27500	0.004994
pges	7.36600	0.247642
pgen	3.92840	0.024239

is no longer the computational bottleneck; it is not necessary to consume more hardware resources.

Chapter 5

Summary and Future Work

In the field of embedded systems research, 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.

5.1 Related to CAL-DIF Project

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 cir-

cuits 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.

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 optimizing thread granularity.

5.2 Related to MPC Project

In this report, we have proposed a general framework for modeling, analyzing, and developing fast parallel implementations of the algorithms used in 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. One example is that the QR algorithm is a better candidate for solving the system of linear equations within Newton-KKT and active set methods. Parallel implementations of the QR algorithm need to be developed. Furthermore, because the communication times are greatly dependent upon the specific hardware the methods described here need to be applied to the 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.

ACKNOWLEDGMENTS

The research work mentioned in the report is supported in part by the Marshall Plan Scholarship Program of Austrian Marshall Plan Foundation.

I'd like to thank my advisor, Professor Shuvra Bhattacharyya for giving me an invaluable opportunity to work on challenging and extremely interesting projects over the past four years.

I would also like to thank all the nice people such as Professor Gabriele Abermann, Professor Gerhard Joechtl, Rosalyn Eder, Simon Kranzer, Bernadette Himmelbauer, Peter Ott and etc in Salzburg University of Applied Sciences for the valuable discussion and wonderful time I spent there.

BIBLIOGRAPHY

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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

[15] "Open rvc cal compiler." [Online]. Available: http://sourceforge.net/apps/trac/orcc/

[16] 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.

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

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

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

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] R. Gu, J. W. 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*, 2009, pp. 565–568.

[27] 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.

[28] 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.

[29] 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

[30] 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.

[31] 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/

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

[39] 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.

[40] 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

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

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

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.

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

[49] 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.

[50] 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

[51] 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.

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

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] ——, "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.

[58] 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.

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

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.