Exploiting Statically Schedulable Regions in Dataflow Programs

Ruirui Gu, Jörn W. Janneck, Mickaël Raulet, Shuvra S. Bhattacharyya

J Sign Process Syst
DOI 10.1007/s11265-009-0445-1

Exploiting Statically Schedulable Regions in Dataflow Programs

Ruirui Gu · Jörn W. Janneck · Mickaël Raulet · Shuvra S. Bhattacharyya

Received: 23 June 2009 / Revised: 16 December 2009 / Accepted: 22 December 2009

Abstract Dataflow descriptions have been used in a wide range of Digital Signal Processing (DSP) applications, such as multi-media processing, and wireless communications. Among various forms of dataflow modeling, Synchronous Dataflow (SDF) is geared towards static scheduling of computational modules, which improves system performance and predictability. However, many DSP applications do not fully conform to the restrictions of SDF modeling. More general dataflow models, such as CAL (Eker and Janneck 2003), have been developed to describe dynamically-structured DSP applications. Such generalized models can express dynamically changing functionality, but lose the powerful static scheduling capabilities provided by SDF. This paper 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 (Eker and Janneck 2003), DIF (Hsu et al. 2005) and CAL2C (Roquier et al. 2008). In addition to detecting SDF-like regions, we apply existing SDF scheduling techniques to exploit the static properties of these regions within enclosing dynamic dataflow models. Furthermore, we propose an optimized approach for mapping SDF-like regions onto parallel processing platforms such as multi-core processors.

Keywords CAL · DIF · Dataflow · Quasi-static scheduling · Multicore processors

R. Gu (✉) · S. S. Bhattacharyya
Department of ECE and UMIACS, University of Maryland, College Park, MD 20742, USA
e-mail: rgu@umd.com

S. S. Bhattacharyya
e-mail: ssb@umd.com

J. W. Janneck
Xilinx Research Labs, San Jose, CA 95124, USA
e-mail: jorn.janneck@xilinx.com

M. Raulet
IETR/INSA Rennes, 35043, Rennes, France
e-mail: mickael.raulet@insa-rennes.fr

1 Introduction

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 [4]. 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 [5]).

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

Published online: 29 January 2010

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 [1] is a language for specifying dataflow actors in a way that is fully general (in terms of expressive power), while clearly exposing functional structures that are useful in detecting important special cases of actor behaviors (e.g., SDF or SDF-like actor behaviors). The CAL language, in terms of its high level of abstraction, is similar to the Stream-Based Functions (SBF) model of computation [6]. 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 [2] 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 [2]. CAL2C [3, 7] 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 [8] and can be downloaded from [9].

In this paper, 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, in this paper 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 paper, we introduce the concept of an *SSR actor*, which is a subsystem within an SSR that can be treated as an SDF actor for purposes of scheduling. In terms of the components in the original CAL specification, an SSR actor may correspond to a single CAL actor or part of (a subset of the functionality within) a CAL actor. Scheduling based on SSR actors is thus of significantly more general applicability compared to conventional SDF scheduling, where SDF actors in the original specification are treated as indivisible "black boxes".

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

A preliminary, partial summary of this paper was presented in [10]. This paper incorporates the following further developments compared to the earlier presentation of [10]. First, we develop a precise and comprehensive formulation of our SSR detection methods in terms of relevant graph-theoretic concepts. Second, we discuss how capabilities of TDP can be exploited in new ways to achieve efficient scheduling of SSRs. We also discuss how integrating our methods for SSR detection and TDP-based scheduling into CAL2C provides capabilities for efficient, automated implementation of video processing systems. Third, we explore novel techniques for mapping CAL networks into multi-core systems by grouping dynamic ports with SSRs into a form of subsystem that we call *weakly connected SSRs*. Our transformation techniques are demonstrated on the IDCT module of an MPEG Reconfigurable Video Coding (RVC) system and an MPEG-4 RVC simple profile (SP) decoder.

This paper is organized as follows. Section 2 introduces previous work related to dataflow models, the

CAL language, and related efforts on extracting SDF-like parts from dynamic dataflow models. Section 3 outlines our methods and notations for translation and analysis across different modeling languages. In Section 4, we introduce the concept of SSRs, and develop a detailed procedure for deriving SSRs from CAL networks. Section 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 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. Conclusions and directions for future work are discussed in Section 7.

2 Related Work

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 [11], Kahn process networks [12], dataflow architectures [13], and dataflow process networks [14]. Synchronous dataflow (SDF) is a specialized form of dataflow that is streamlined for efficient representation of DSP systems [4].

Since the introduction of SDF, a variety of such *DSP-oriented dataflow models of computation* have been proposed, and DSP-oriented models have been incorporated into many commercial design tools, including Agilent ADS, Cadence SPW (later acquired by CoWare), National Instruments LabVIEW, and Synopsys CoCentric. Useful relationships between dataflow and synchronous languages have also been developed, which helps to connect DSP-oriented dataflow methods to other popular tools, such as Simulink by The MathWorks (e.g., see [15]). 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 [16]) .

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

A simple example is illustrated in Fig. 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 Fig. 1 conforms to the SDF model.

A limitation of SDF and related models, such as cyclo-static dataflow [17] and homogeneous SDF (HSDF) [4], 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

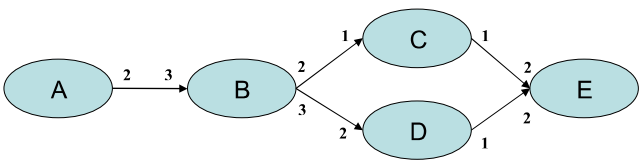

Figure 1 A simple example of a dataflow (SDF) model.

flow model [18], stream-based functions [6], enable-invoke dataflow (EIDF) [19], and the CAL actor language [1].

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 [20]. The DIF package (TDP) [2, 19] 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 [20].

TDL is designed as a standard approach for specifying DSP-oriented dataflow graphs at a high level of abstraction that is suitable for both programming and interchange. TDL provides a unique set of semantic features for specifying graph topologies, hierarchical design structure, dataflow-related design properties, and actor-specific information. TDP accompanies TDL, and provides a variety of intermediate representations, analysis techniques, and graph transformations that are useful for working with dataflow graphs that have been captured by TDL. Mocgraph is a companion tool that is provided along with TDP. Mocgraph can be viewed as a library of algorithms and representations for working with generic graphs, whereas TDP is a specialized package for working with dataflow graphs. For more details on TDL, TDP, and Mocgraph, we refer the reader to [2, 19].

For example, TDP includes a transformation that converts SDF representations into equivalent homogeneous SDF (HSDF) representations based on the algorithm introduced in [4]. Such a transformation can in general expose additional concurrency [21] that is not represented explicitly in the original SDF graph. In this paper, 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 paper, such extraction and exploitation of SSRs provides a powerful new methodology for optimized implementation of dataflow graphs. In comparison, [21] presents in-depth dataflow based analysis and exploitation of parallelism in the design and implementation of an MPEG RVC decoder, while this paper focuses on detailed description of the SSR detection algorithm.

Compared to other design tools for representation and transformation of dataflow graphs—such as SysteMoC [22], PeaCE [23], and stream-based functions [6]—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.

2.3 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) [24], uses the CAL actor language [1] for specifying functional components, and dataflow as the composition formalism [25].

An integrated set of tools related to CAL are presented in OpenDF [26]. 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 runtime, 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 [27] sketches a method to classify CAL actors into different dataflow classes for efficient scheduling. Boutellier et al. [28] propose an approach to quasi-static multiprocessor scheduling of CAL-based RVC applications. The approach involves the dynamic selection and execution of "piecewise static schedules" based on novel extensions of flow shop scheduling techniques.

Many previous research efforts have focused on task mapping for multiprocessor systems from other kinds of specification models or languages (e.g., see [16]). For example, Li et al. [29] 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. [30] 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 paper are the capability to decompose CAL actors based on their formal action- and port-based semantics, and to construct and subsequently transform SSRs and SSR actors from these decomposed representations. As a result, our methodology has access to and is capable of exploiting the detailed formal modeling semantics of the CAL language, which includes formal modeling of both communication between actors, as well as computations and state transitions within actors. Additionally, our methods provide a novel framework of quasi-static scheduling in terms of SSR actors.

3 Analysis Framework

Our method to optimize implementation of DSP applications combines the advantages of three complementary tools, as shown in Fig. 2. The given DSP application is initially described as a CAL network that is composed of CAL actors. The CAL-based dataflow representation is then translated into a DIF-based intermediate representation for analysis by TDP. This TDP-driven analysis produces a set of SSRs, and an associated quasi-static schedule, which is then translated into a reformulated CAL specification. This transformed CAL code is then translated to a C code implementation using CAL2C. The generated CAL2C implementation is optimized to exploit the static structure provided by the SSRs and their enclosing quasi-static schedule.

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

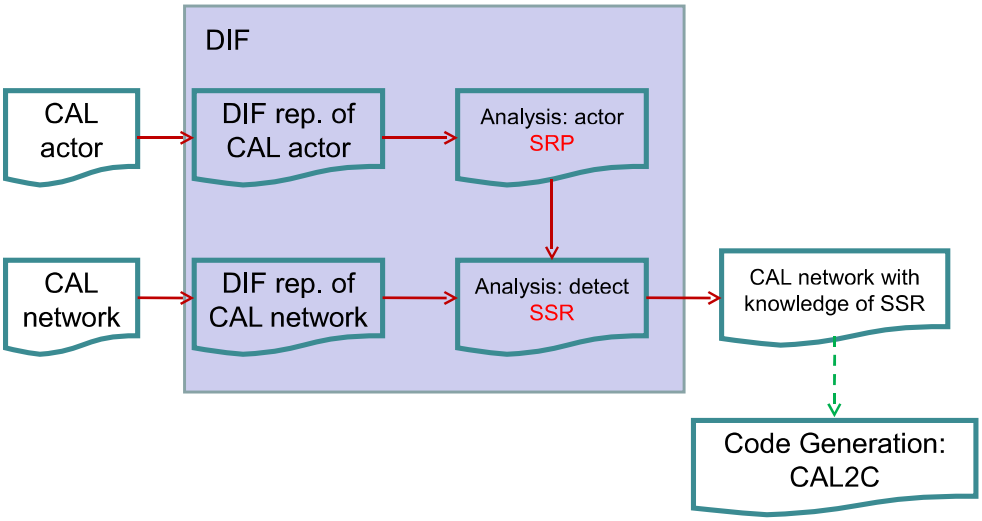

Figure 2 Outline of method for optimizing dataflow graph implementation.

$l \succ m$ 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.

4 Derivation of Statically Schedulable Regions

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

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

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

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

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

Partitioning across ports from different actors is based on connections in the enclosing CAL network. If ports of distinct actors are connected in the CAL network, then they are combined into the same partition, including any other subsets of ports within the same actors that satisfy coupling relationships with respect to the ports.

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

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

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

Figure 3 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 $\{B.c, B.y\}$.

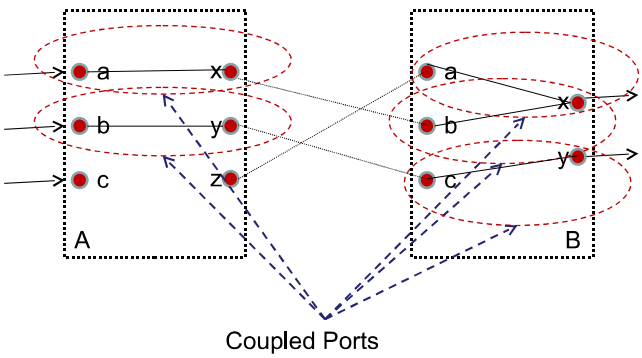

Figure 3 An illustration of coupled ports and CRGs.

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 [31]). 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 4 shows an example of a directed graph, the undirected version of that graph, the associated weakly connected components.

Figure 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 Fig. 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. 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.

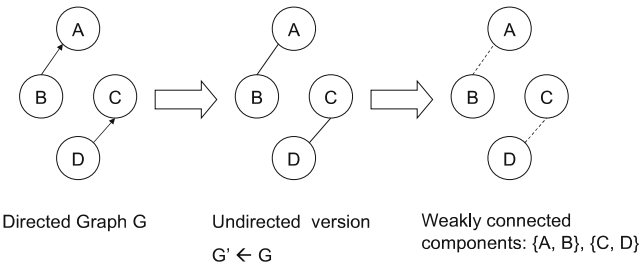

Figure 4 An illustration of weakly connected components.

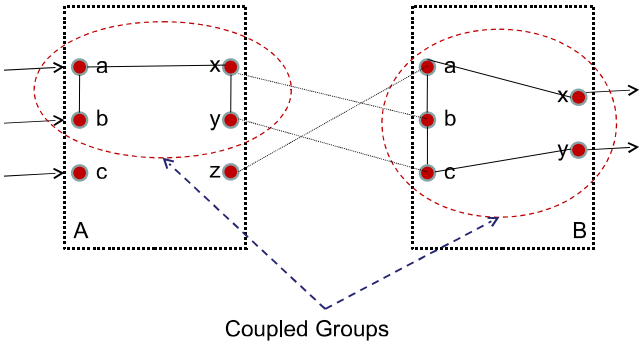

Figure 5 An illustration of coupled groups.

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 coupled group $R = a_1, a_2, \ldots, a_n$, we construct the static relationship graph of R by first instantiating a vertex x_{a_i} for each $a_i \in R$ such that a_i is a static rate port, and a vertex v_z for every action z in the actor. We then instantiate an edge (x_{a_i}, v_z) for every ordered pair (a_i, z) such that $a_i \in ports(z)$. By definition, the static relationship graph is a bipartite graph. Figure 6 shows an example of a static relationship graph and the statically-related groups derived from Fig. 5.

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

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

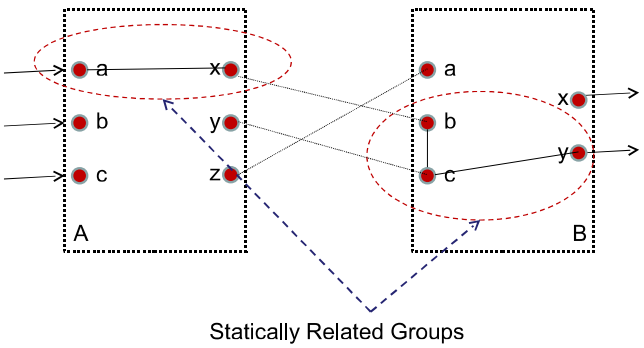

Figure 6 An illustration of statically-related groups.

$p_1 \in A_1$, $p_2 \in A_2$, 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

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

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

$$r_i = \bigcup_{j=1}^{m_i} x_{i,j}. \tag{2}$$

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

Figure 7 shows an example of an SRG graph and the obtained statically schedulable region.

5 Scheduling of SSRs

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

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.

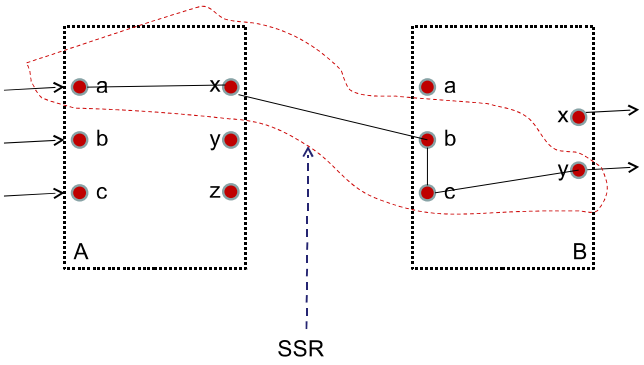

Figure 7 An illustration of a statically schedulable region.

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.

5.1 IDCT Example

Figure 8 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.

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 Fig. 8. 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 [32, 33], 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 Fig. 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 [34].

In the schedule tree shown in Fig. 9, SSR actors that are labeled with purely alphabetic names (no number in the name), such as *tran* and *row*, indicate SSR actors that correspond to the entire computation of the associated CAL actor. On the other hand, SSR actors whose names contain numbers correspond to actors in the CAL network that map to multiple SSR actors across multiple SSRs.

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

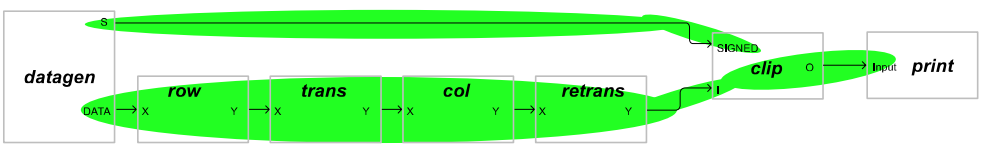

Figure 8 SSRs in the IDCT subsystem.

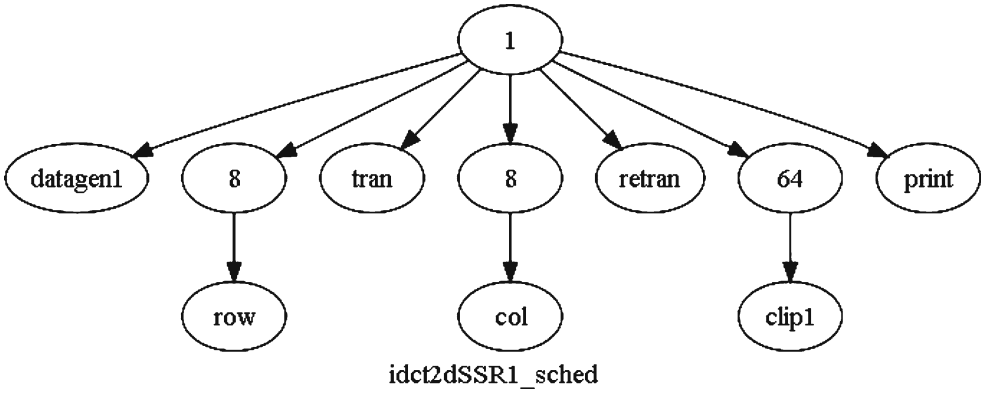

Figure 9 Schedule tree for an SSR in the IDCT example.

5.2 Simulation Results

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

The second version (V2) uses CAL2C-SSR. This version exploits the SSRs illustrated in Fig. 8, 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 Fig. 10, the *IDCT* system becomes an SDF model that runs as a single thread. Since entirely static scheduling is used in this version, V3 is the most efficient in terms of execution speed.

We experimented with all three *IDCT* versions using Microsoft Visual Studio. The results are shown in Fig. 11. Here, V2 shows an improvement in performance of 1.5 times compared to V1, whereas V3 shows the best performance among all three versions.

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

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

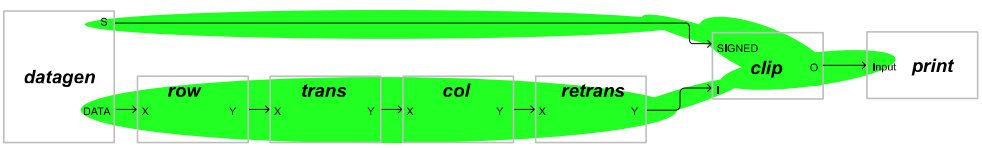

Figure 10 IDCT subsystem with a single SSR.

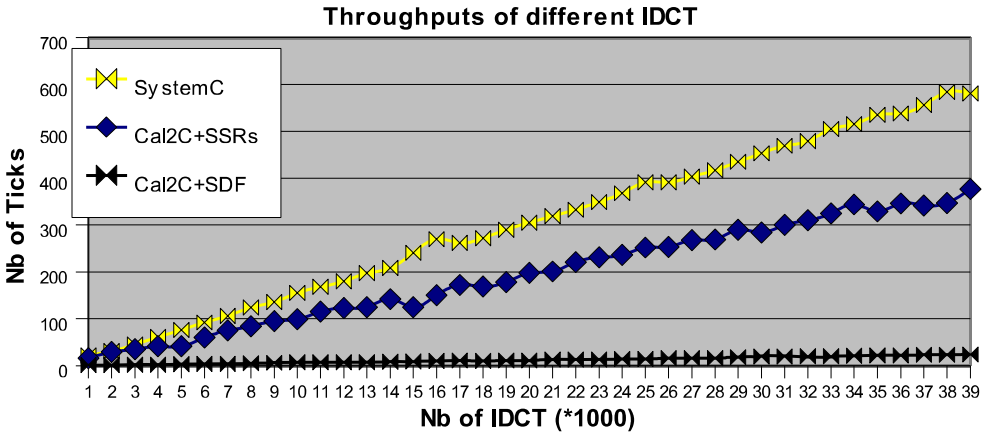

Figure 11 Results: clock cycles vs number of iterations.

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 Fig. 12. When applied to this system, our tools for SSR detection derived a total of 30 SSRs. 32 ports are left outside the SSRs—these are the dynamic ports. By applying our method of strength-based, iterative grouping, we partitioned the 32 dynamic ports across the set of available SSRs. We then used the resulting partitioning result to derive a quasi-static schedule for the system.

For these experiments, we further modified the scheduler in CAL2C [3] 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 [3]. 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

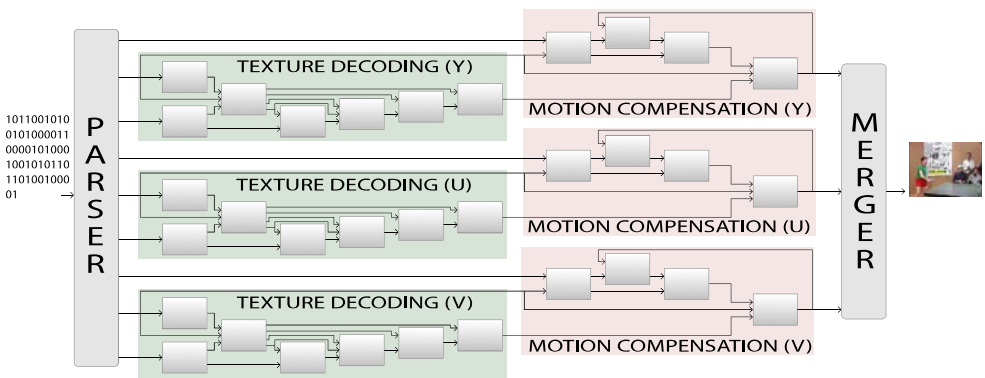

Figure 12 A block diagram of an MPEG RVC decoder.

Table 1 MPEG-4 SP decoder performance for 352 × 288 CIF sequence.

SP decoder	Speed (frame/s)
Monoprocessor with SystemC scheduler	8
Monoprocessor with round robin scheduler	42
Monoprocessor with round robin scheduler and SSRs	44
Dualcore processor with round robin scheduler and SBIG	50

this can lead to poor performance, especially if many actors with low granularity are present.

Compared to a direct translation in SystemC [3], 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 both CIF sequences with size 352 × 288 and sequences with size 624 × 352. A CIF-size image (352 × 288) corresponds to 22 × 18 macroblocks. As shown in Tables 1 and 2, the experimental results demonstrate that CAL2C with quasi-static scheduling using strength-based, iterative grouping (SBIG) on the round robin scheduler has the best performance in a multi-core system. CAL2C with SBIG can be applied to more applications besides MPEG, and this is a useful direction for future work.

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

Table 2 MPEG-4 SP decoder performance for 624 × 352 sequence.

SP decoder (with round robin scheduler)	Speed (frame/s)
Monoprocessor	10
Monoprocessor with SSRs	11
Dualcore processor	15
Dualcore processor with SBIG	16

7 Conclusions

In this paper, 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.

SSRs, with their amenability to static scheduling, not only benefit sequential processing systems, but also help to improve the use of parallel processing platforms, such as multi-core processors. The concept of strength-based, iterative grouping is proposed to carefully integrate scheduling decisions across dynamic ports and SSRs. Using this approach, we demonstrate new techniques for optimized, quasi-static scheduling of CAL networks on multi-core processors.

References

1. Eker, J., & Janneck, J. W. (2003). *CAL language report, language version 1.0—document edition 1*. Electronics Research Laboratory, University of California at Berkeley, Tech. Rep. UCB/ERL M03/48.
2. Hsu, C., Ko, M., & Bhattacharyya, S. S. (2005). Software synthesis from the dataflow interchange format. In *Proceedings of the international workshop on software and compilers for embedded systems* (pp. 37–49). Dallas, Texas.
3. Roquier, G., Wipliez, M., Raulet, M., Janneck, J. W., Miller, I. D., & Parlour, D. B. (2008). Automatic software synthesis of dataflow program: An MPEG-4 simple profile decoder case study. In *Proceedings of the IEEE workshop on signal processing systems*.
4. Lee, E. A., & Messerschmitt, D. G. (1987). Synchronous dataflow. *Proceedings of the IEEE, 75*(9), 1235–1245.
5. Bhattacharyya, S. S., Leupers, R., & Marwedel, P. (2000). Software synthesis and code generation for DSP. *IEEE transactions on circuits and systems—II: Analog and digital signal processing, 47*(9), 849–875.
6. Kienhuis, B., & Deprettere, E. F. (2003). Modeling stream-based applications using the SBF model of computation. *Journal of Signal Processing Systems, 34*(3), 291–300.
7. Wipliez, M., Roquier, G., Raulet, M., Nezan, J.-F., & Deforges, O. (2008). 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*.
8. Wipliez, M., Roquier, G., & Nezan, J. (2009). Software code generation for the RVC-CAL language. *Journal of Signal Processing Systems*. doi:10.1007/s11265-009-0390-z.
9. Open RVC CAL compiler (2009). http://sourceforge.net/apps/trac/orcc/.
10. Gu, R., Janneck, J., Raulet, M., & Bhattacharyya, S. S. (2009). Exploiting statically schedulable regions in dataflow programs. In *Proceedings of the international conference on acoustics, speech, and signal processing* (pp. 565–568). Taipei, Taiwan.
11. Karp, R. M., & Miller, R. E. (1966). Properties of a model for parallel computations: Determinacy, termination, queuing. *SIAM Journal of Applied Math, 14*(6), 1390–1411.
12. Kahn, G. (1974). The semantics of a simple language for parallel programming. In *Proceedings of the IFIP congress*.
13. Dennis, J. B. (1975). *First version of a data flow procedure language*. Tech. Rep., Laboratory for Computer Science, Massachusetts Institute of Technology.
14. Lee, E. A., & Parks, T. M. (1995). Dataflow process networks. *Proceedings of the IEEE*, *83*, 773–799.
15. Lublinerman, R., & Tripakis, S. (2008). Translating data flow to synchronous block diagrams. In *Proceedings of the IEEE workshop on embedded systems for real-time multimedia*.
16. Sriram, S., & Bhattacharyya, S. S. (2009). *Embedded multiprocessors: Scheduling and synchronization* (2nd ed.). Boca Raton: CRC.
17. Bilsen, G., Engels, M., Lauwereins, R., & Peperstraete, J. A. (1996). Cyclo-static dataflow. *IEEE Transactions on Signal Processing, 44*(2), 397–408.
18. Buck, J. T., & Lee, E. A. (1993). The token flow model. In L. Bic, G. Gao, & J. Gaudiot (Eds.), *Advanced topics in dataflow computing and multithreading*. Los Alamitos: IEEE Computer Society.
19. Plishker, W., Sane, N., Kiemb, M., Anand, K., & Bhattacharyya, S. S. (2008). Functional DIF for rapid prototyping. In *Proceedings of the international symposium on rapid system prototyping* (pp. 17–23). Monterey, California.
20. Hsu, C., & Bhattacharyya, S. S. (2005). Porting DSP applications across design tools using the dataflow interchange format. In *Proceedings of the international workshop on rapid system prototyping* (pp. 40–46). Montreal, Canada.
21. Gu, R., Janneck, J. W., Bhattacharyya, S. S., Raulet, M., Wipliez, M., & Plishker, W. (2009). Exploring the concurrency of an MPEG RVC decoder based on dataflow program analysis. *IEEE Transactions on Circuits and Systems for Video Technology, 19*, 1646–1657.
22. Haubelt, C., Falk, J., Keinert, J., Schlichter, T., Streubühr, M., Deyhle, A., et al. (2007). A SystemC-based design methodology for digital signal processing systems. *EURASIP Journal on Embedded Systems, 2007*, Article ID 47580 (22 pp.).
23. Sung, W., Oh, M., Im, C., & Ha, S. (1997). Demonstration of hardware software codesign workflow in PeaCE. In *Proceedings of the international conference on VLSI and CAD*.
24. Bhattacharyya, S. S., Eker, J., Janneck, J. W., Lucarz, C., Mattavelli, M., & Raulet, M. (2009). Overview of the MPEG reconfigurable video coding framework. *Journal of Signal Processing Systems*. doi:10.1007/s11265-009-0399-3.
25. Lucarz, C., Mattavelli, M., Thomas-Kerr, J., & Janneck, J. (2007). Reconfigurable media coding: A new specification model for multimedia coders. In *Proceedings of the IEEE workshop on signal processing systems*.
26. Bhattacharyya, S. S., Brebner, G., Eker, J., Janneck, J. W., Mattavelli, M., von Platen, C., et al. (2008). OpenDF—a dataflow toolset for reconfigurable hardware and multicore systems. *ACM SIGARCH Computer Architecture News, 36*(5). http://hal.archives-ouvertes.fr/hal-00398827/en/.
27. Platen, C. V., & Eker, J. (2008). Efficient realization of a CAL video decoder on a mobile terminal. In *Proceedings of the IEEE workshop on signal processing systems*.
28. Boutellier, J., Sadhanala, V., Lucarz, C., Brisk, P., & Mattavelli, M. (2008). Scheduling of dataflow models within the reconfigurable video coding framework. In *Proceedings of the IEEE workshop on signal processing systems*.
29. Li, M., Wang, H., & Li, P. (2003). Tasks mapping in multicore based system: Hybrid ACO&GA approach. In *Proceedings of the international conference on ASIC*.
30. Ennals, R., Sharp, R., & Mycroft, A. (2005). Task partitioning for multi-core network processors. In *Proceedings of the international conference on compiler construction*.
31. Cormen, T. H., Stein, C., Leiserson, C. E., & Rivest, R. L. (2001). *Introduction to algorithms* (2nd ed.). Cambridge: MIT.
32. Bhattacharyya, S. S., Murthy, P. K., & Lee, E. A. (1997). APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations. *Journal of Design Automation for Embedded Systems, 2*(1), 33–60.
33. Plishker, W., Sane, N., & Bhattacharyya, S. S. (2009). A generalized scheduling approach for dynamic dataflow applications. In *Proceedings of the design, automation and test in Europe conference and exhibition* (pp. 111–116). Nice, France.
34. Ko, M., Zissulescu, C., Puthenpurayil, S., Bhattacharyya, S. S., Kienhuis, B., & Deprettere, E. (2007). Parameterized looped schedules for compact representation of execution sequences in DSP hardware and software implementation. *IEEE Transactions on Signal Processing, 55*(6), 3126–3138.

Ruirui Gu is currently a Ph.D. student in the Department of Electrical and Computer Engineering University of Maryland at College Park. Her research interests include signal processing systems, embedded software, hardware/software co-design, parallel computing systems, architectures and software. She has worked in a variety of applications, including audio and video coding and processing, advanced control systems, reconfigurable computing and wireless communications. She received the B.S. degree and M.S. degree from Shanghai Jiao Tong University.

Jörn W. Janneck graduated from the University of Bremen in 1995, worked for the Fraunhofer Institute for Material Flow and Logistics in 1996, and then pursued graduate studies at the ETH Zurich, where he received his Dr. sc. techn. in 2000. From 2000 to 2003 he worked as a visiting scholar at the University of California at Berkeley as a member of the Ptolemy group. In 2003 he joined the Xilinx Research Labs, where he focuses on high level programming methodologies for FPGAs, and in particular on dataflow. His research interests include concurrency, programming languages, compilers, and the engineering of parallel computing systems. He has worked in a variety of application areas, including material flow modeling and simulation, image and video coding and processing, wireless communications, distributed algorithms and simulation, and discrete-event modeling of complex electro-mechanical systems.

Shuvra S. Bhattacharyya is a Professor in the Department of Electrical and Computer Engineering University of Maryland at College Park. He holds a joint appointment in the University of Maryland Institute for Advanced Computer Studies (UMIACS), and an affiliate appointment in the Department of Computer Science. Dr. Bhattacharyya is coauthor or coeditor of five books and the author or coauthor of more than 150 refereed technical articles. His research interests include signal processing systems, architectures, and software; biomedical circuits and systems; embedded software; and hardware/software co-design. He received the B.S. degree from the University of Wisconsin at Madison, and the Ph.D. degree from the University of California at Berkeley. Dr. Bhattacharyya has held industrial positions as a Researcher at the Hitachi America Semiconductor Research Laboratory (San Jose, California), and Compiler Developer at Kuck & Associates (Champaign, Illinois).

Mickaël Raulet received his postgraduate certificate in signal, telecommunications, images, and radar sciences from Rennes University in 2002, and his Engineering degree in electronic and computer engineering from National Institute of Applied Sciences (INSA), Rennes Scientific and Technical University. Next in 2006, he received a Ph.D. degree from INSA in electronic and signal processing in collaboration with the software radio team of Mitsubishi Electric ITE (Rennes - France). He is currently in the Institute of Electronics and Telecommunications of Rennes (IETR) where he is a research engineer in rapid prototyping of standard video compression on embedded architectures (multi DSP architecture). Since 2007, he is involved in the ISO/IEC JTC1/SC29/WG11 standardization activities (better known as MPEG) such as a Reconfigurable Video Coding Expert.