Exploring the concurrency of an MPEG RVC decoder based on dataflow program analysis

Ruirui Gu, Jörn W. Janneck, Shuvra S. Bhattacharyya, Mickaël Raulet, Matthieu Wipliez, William Plishker

Exploring the Concurrency of an MPEG RVC Decoder Based on Dataflow Program Analysis

Ruirui Gu*, Jörn W. Janneck‡, Shuvra S. Bhattacharyya*,
Mickaël Raulet†, Matthieu Wipliez†, William Plishker*
*Electrical and Computer Engineering, University of Maryland at College park (U.S.A)
†IETR Laboratory - UMR CNRS 6164 - Rennes (France)
‡Xilinx Inc. - San Jose (U.S.A)

***Abstract*—This paper presents an in-depth case study on dataflow-based analysis and exploitation of parallelism in the design and implementation of an MPEG RVC (reconfigurable video coding) decoder. Dataflow descriptions have been used in a wide range of digital signal processing (DSP) applications, such as applications for multimedia processing and wireless communications. 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. In this paper, we use the CAL language as a concrete framework for representing and demonstrating dataflow design techniques.**

Furthermore, we also describe our application of The DIF package (TDP), a software tool for analyzing dataflow networks, to the systematic exploitation of concurrency in CAL networks that are targeted to multi-core platforms. Using TDP, one is able to automatically process regions that are extracted from the original network, and exhibit properties similar to synchronous dataflow (SDF) models. This is important in our context because powerful techniques, based on static scheduling, are available for exploiting concurrency in SDF descriptions. Detection of SDF-like regions is an important step for applying static scheduling techniques within a dynamic dataflow framework. Furthermore, segmenting a system into SDF-like regions also allows us to explore cross-actor concurrency that results from dynamic dependencies among different regions. Using SDF-like region detection as a preprocessing step to software synthesis generally provides an efficient way for mapping tasks to multi-core systems, and improves the system performance of video processing applications on multi-core platforms. [1]

***Index Terms*—MPEG RVC, dataflow, CAL, DIF, concurrency, parallel processing.**

I. Introduction

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 [1]. 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 [2]). 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 paper, and are useful directions for future work.

References [1], [3], [4] cover related aspects of reconfigurable video coding and CAL-oriented tools. In particular, [1] gives an overview of the overall RVC framework; Reference [3] provides details on the software code generator CAL2C; and Reference [4] elaborates on a hardware code generator for CAL. In contrast, this paper is distinctive in it focus on analyzing concurrency and exploiting parallelism; the topic

of concurrency is not addressed in depth in References [1], [3] and [4].

Using CAL as a concrete design representation framework, this paper places emphasis on answering the last two questions described above. More specifically, this paper 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 II 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 III analyzes inter-actor concurrency obtained from CAL specifications from the viewpoint of both hardware and software implementation. Section IV proposes techniques for cross actor-optimization that enhance multi-core system performance. Simulation results are also presented in this section. Conclusions and future work are discussed in section V.

II. Background

A. 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 [1]. 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 [5] and MPEG-C part 4 [6] 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.

B. 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 [7], Kahn process networks [8], and dataflow architectures [9]. 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 simple example is illustrated in Figure 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 1 conforms to the SDF model.

A limitation of SDF and related models, such as cyclo-static [11] and single-rate [12] 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 [13], functional DIF [14], and the CAL actor language [15] that is targeted in this paper.

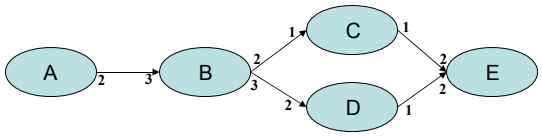


Fig. 1. A simple example of a dataflow (SDF) model.

C. 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 [16]. 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.

D. 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 [15] as the language for specifying FUs, and the FU network language for the dataflow composition [6]. 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 [17]. 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.

E. 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 [18]). 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 paper addresses the systematic mapping onto parallel platforms of concurrent tasks that are extracted from CAL programs.

III. Inter-actor concurrency analysis

A. 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 paper 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 2. In Figure 2, the shaded area indicated as *texture decoding* represents a super actor that is described in FNL . Similarly, the shaded area labeled as *motion compensation* also represents a hierarchical actor in our design. Furthermore, inside the actor *texture decoding*, the *Inverse DCT* actor represents a lower-level super actor, which is also described in FNL and is composed of several *atomic* (non-hierarchical) actors/FUs. The other blocks in the diagram are atomic actors/FUs.

Overall, in the MPEG-4 SP decoder shown in Figure 2, there are three hierarchies and atomic actors and super actors from different hierarchies are interleaved. Note that for readability, only one edge is shown in cases where two actors are connected by more than one edge. It is possible, for example, that multiple edges connect the same pair of actors because of connections between different interfaces of hierarchical subsystems.

B. Data parallelism inside CAL networks

In Figure 2, there are three sub-systems that handle Y, U and V separately. These three sub-systems share the same set of processing modules in the form of CAL actors that differ only in their associated sample rates.

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

The subsystems for Y, U and V are concurrent in the sense that they handle signals from different channels. These signals are generated by the *parser* actor, and then are directed to the Y, U, and V subsystems for processing. In this way, the CAL network explicitly exposes inter-actor, and inter-subsystem concurrency in the overall application.

C. 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 paper.

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 [19]). 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.

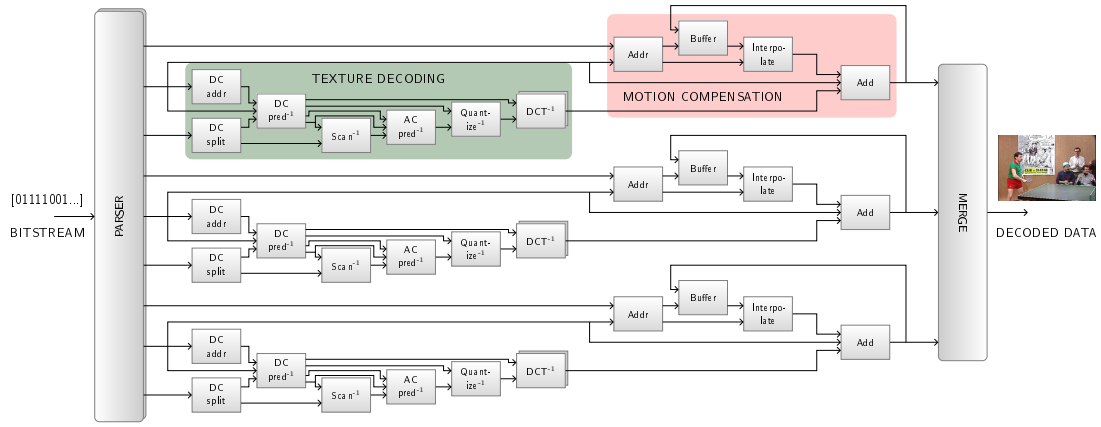


Fig. 2. An RVC block diagram of an MPEG-4 Simple Profile decoder.

D. 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 [4]. 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 paper; we refer the reader to [4] 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 [20], [21] 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.

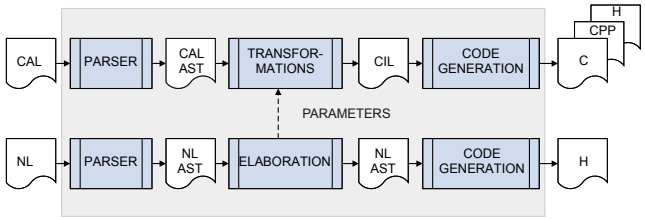


Fig. 3. CAL2C compilation process: The action translation process starts with an abstract syntax tree (AST) derived from the CAL source code; the transformed CAL AST is expressed in the C intermediate language (CIL) [22], where CAL functional constructs are replaced by imperative ones.

In CAL2C, software synthesis from a CAL network includes two parts: actor transformation [20] and network transformation [21]. Inside an actor, CAL translation is performed 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 [20]. 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 illustrates how CAL2C works. For further details on CAL2C, we refer the reader to [21].

In [21], we have applied CAL2C successfully on our CAL-based design for the MPEG-4 SP decoder. Simulation results show that the synthesized C-software is as fast as 20 frames/s, which provides near-real-time performance for the QCIF format (25 frames/s) on a standard PC platform. It is interesting to note that our CAL-based speed processing generated from CAL2C is scalable in terms with the number of macro-blocks decoded per second (MB/s) (the number of MB/s remains constant when dealing with larger image sizes). Furthermore, this number can be increased if we use more powerful processors.

Although both forms of design produce code in the same kind of language, code generated from CAL2C is different compared to implementations that use C/C++ as the starting point. As a dataflow language, CAL restricts the way in which designers can describe applications, and these restrictions carry over through CAL2C to produce code that is more modular and purely dataflow-oriented compared to implementations that are developed directly from C/C++. This is illustrated in Figure 4.

After obtaining a set of threads from CAL2C, the mapping of these threads onto the targeted multi-core platform remains

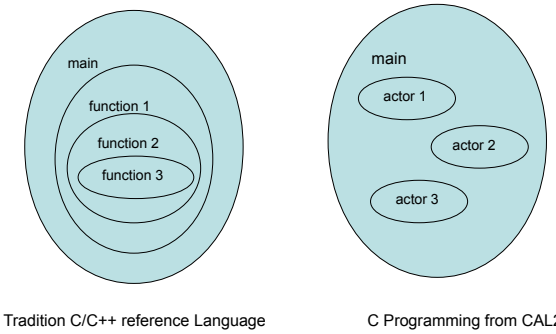


Fig. 4. Comparison between direct-C/C++-based implementation and implementation using CAL2C.

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.

IV. 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.

A. DIF and network analysis capability

In this section, we present our application of the dataflow interchange format (DIF) package [2], [12], 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 [23]. For the complete DIF language grammar and a detailed syntax description, we refer the reader to [12].

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 paper, 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 paper, 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 IV-C, we develop the concept of SSRs in details.

Compared to other design tools for representation and transformation of dataflow graphs — such as SysteMoC [24], PeaCE [25], and stream-based functions [13] — 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.

B. Interface between DIF and CAL

Our method to optimize implementation of DSP applications combines the advantages of three complementary tools, as shown in Figure 5. The given DSP application is initially described as a CAL network, which is a highly expressive form of dataflow graph. The CAL-based dataflow representation is then translated into a DIF-based intermediate representation for analysis by TDP. This TDP-driven analysis produces a set of SSRs, and an associated quasi-static schedule, which is then translated into a reformulated CAL specification. This transformed CAL code is then translated to a C code implementation using CAL2C. The generated CAL2C implementation is optimized to exploit the static structures provided by the SSRs and their enclosing quasi-static schedules.

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

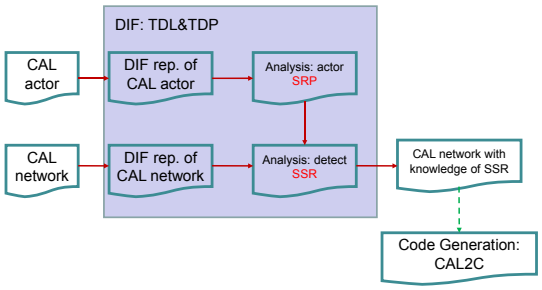


Fig. 5. Overview of our CAL- and DIF-based method for optimizing dataflow graph implementation. SRP represents statically related port and SSR represents statically related region.

Describing an actor in CAL involves describing not only its ports, but also the structure of its internal state; the actions it can perform; what these actions do (such as token production and token consumption, and updating of actor state); and how to determine the action that the actor will perform next. When performing network dataflow analysis, we analyze interactions among ports, state variables, and guard conditions of CAL actors. In our current research, which focuses on deriving and utilizing information about the token production and consumption rates of actors, action priority is not taken into consideration. This is because action priority only affects the order of action execution within individual actors; it does not affect the numbers of tokens that are produced or consumed.

C. 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 paper, 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 connectivity 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 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. 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 6, there are four stages in our application of the PCG: coupled ports (CPs), coupled groups (CGs), statically related groups (SRGs), and statically schedulable regions (SSRs). Using TDP, we repeatedly apply two key techniques when working with the PCG — techniques of partitioning and grouping — through the connected component analysis of the PCG. Transformation of PCG is the procedure of all the ports in the CAL network going through the above four stages. The detailed description on strong connections, statically schedulable regions and PCG derived in our design flow is the result of network analysis in TDP [26].

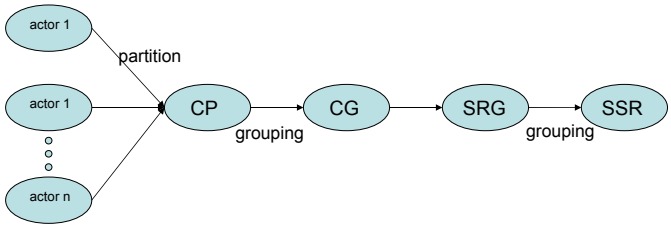


Fig. 6. SSR detection in PCG.

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

D. 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 7 illustrates SSRs within the IDCT subsystem. Here, the main body of the IDCT is composed of the actors *row*, *tran*, *col*, *retran* and *clip*. The *dataGen* and *print* actors are used to complete a testbench for the network — $dataGen$ is responsible for generating input data, and $print$ for displaying the output from the IDCT computation. The shaded regions shown in the figure correspond to the different SSRs, which are unique to the application.

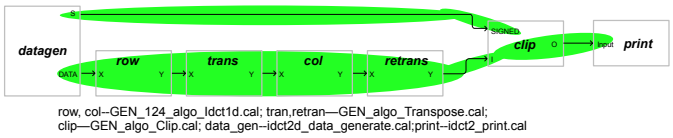


Fig. 7. SSRs in the IDCT subsystem.

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 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.

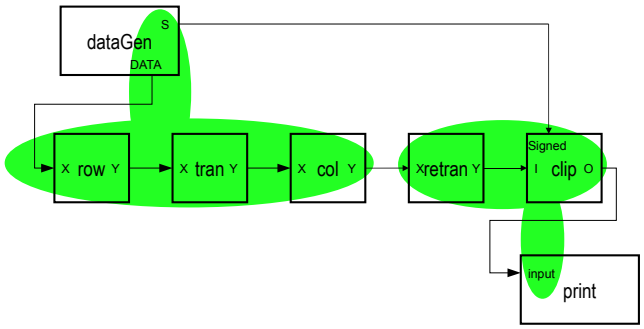
In another word, SSR distinguish weak connections from strong connections. Thus, when two SSRs are allocated onto two cores, the connections for the SSRs between the cores are weak. In our example, semaphores can be used for the two cores to access the same data. In an SSR-based multi-threaded system, 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 8. Four CAL actors are mapped into one core, and the other three actor are mapped into the other core. There are other possible options with differences in the numbers of actors that are mapped to individual cores. Whatever option is used for mapping actors, although inter-actor concurrency is maintained, for each macroblock processed by the IDCT module, execution of actors is sequential. Furthermore, since there are two paths between actors *dataGen* and *clip*, as shown in Figure 8, if these two actors are mapped onto separate cores, there is a relatively large amount of data communication between the cores, which in turn results in a large amount of context switch overhead on the individual cores.

If we map the IDCT onto a dual-core system based on SSR analysis, a straightforward mapping for this case is shown in Figure 7. In this case, the connections between the cores are weak connections inside both the *dataGen* and *clip* actors. These weak connections can be implemented using semaphore primitives. Furthermore, inside each core, the actions can be statically scheduled in terms of checks on an appropriately defined semaphore. Here we can easily take advantage of well known SDF scheduling techniques, such as APGAN [27] [28]. An example of scheduling of SSRs, including the actor *clip*, is shown in Figure 9.



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

Fig. 8. Actor-level mapping onto a multi-core platform.

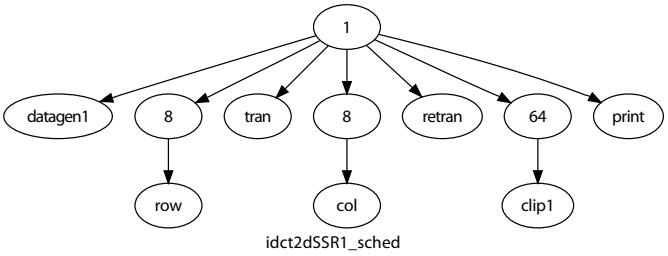
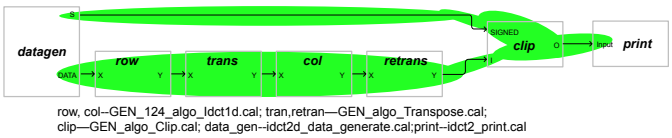


Fig. 9. Scheduling tree for one SSR in the IDCT.

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 8.

The second version (V2) uses CAL2C-SSR. This version exploits the SSRs illustrated in Figure 7, and employs a quasi-static integration of static schedules for these SSRs with top-level dynamic scheduling. In this version, two SSRs are mapped onto two cores, and semaphore primitives are used for inter-SSR communication.

The third version (V3) also uses CAL2C-SSR. This version also uses a modified, more predictable version of the *clip* actor that can be used when the input data is known in advance. In the new version of *clip*, the ports *Signed* and *O* are rewritten to become coupled ports. Then the original two SSRs are combined as one SSR through connections inside *clip*. In the illustration of V3 shown in Figure 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.



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--idct2_print.cal

Fig. 10. IDCT subsystem with one SSR.

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

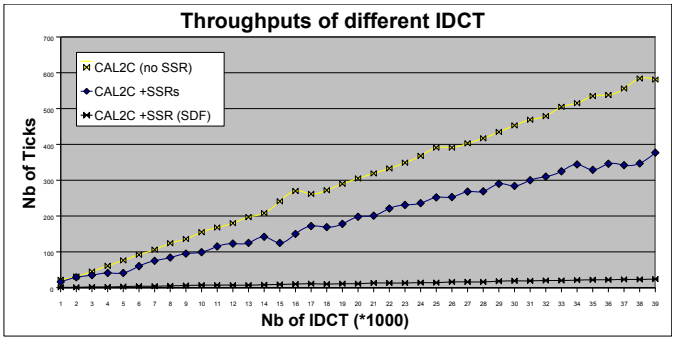


Fig. 11. Results: clock cycles vs number of iterations.

E. Concurrency analysis of the MPEG-4 SP decoder

When we analyze the MPEG-4 SP decoder in Figure 2 in the domain of TDP, the first step is to translate the hierarchical system into a flattened one in which every actor is an atomic actor.

In TDP, a *fork* actor is introduced to implement dataflow-style broadcasting when needed (i.e., when data must be copied to multiple outgoing edges). For example, *Header* is an atomic actor inside the super actor *parser* in the CAL network of Figure 2, and the tokens produced from the *BTYPE* port of the actor *Header* are broadcast to five different input ports of different actors. Thus, in the intermediate representation derived by TDP, a fork actor *GEN-mgnt-fork* is inserted between *Header* and the five actors that are destinations of the broadcast. Conceptually, whenever *GEN-mgnt-fork* fires,

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 I
MPEG-4 SP DECODER PERFORMANCE.

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 paper.

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 [3]. 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 [29] executing each actor in a loop; an actor is fired until input tokens are available and output FIFOs are not full. SSRs can be easily incorporated in this fully software-based implementation, independent from SystemC, by removing all of the possible tests on the FIFOs when an SSR is detected.

We conducted experiments involving the application of CIF sequences with size 352x288. A CIF-size image (352x288) corresponds to 22x18 macroblocks. As shown in Table I, 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 [30]. Further exploration on this class of "granularity-adjustment" transformations for SSRs is a useful direction for further work.

V. CONCLUSIONS

This paper describes an approach based on dataflow representations to coping with the growing complexity of video processing algorithms. We demonstrate this approach on an in-depth case study involving the design of an MPEG reconfigurable video coding (RVC) description of the MPEG-4 simple profile decoder. The system is originally represented using an actor-oriented dataflow language called CAL. Code generators, such as CAL2C, that translate CAL into C code are then described, and this is followed by an analysis of inter-actor concurrency in CAL-based dataflow representations. Next, we describe an approach for automatically detecting and exploiting structures called statically schedulable regions (SSRs) from within CAL networks. We then show how SSRs can be used to significantly improve the efficiency and predictability of multi-core video processing systems.

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

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.

REFERENCES

[1] 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

[2] 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.

[3] 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

[4] 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

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

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

[7] 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.

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

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

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

[11] 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.

[12] C. Hsu, I. Corretjer, M. Ko., W. Plishker, and S. S. Bhattacharyya, "Dataflow interchange format: Language reference for DIF language version 1.0, users 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] E. Gagnon, "Sablecc: An object-oriented compiler framework," School of Computer Science, McGill University, Montreal, Canada, Tech. Rep., 1998.

[24] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubhr, 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.

[25] 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.

[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.

[27] 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.

[28] 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.

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

[30] 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.