EXPLOITING STATICALLY SCHEDULABLE REGIONS IN DATAFLOW PROGRAMS

Ruirui Gu, Jorn W. Janneck, Mickael Raulet, Shuvra S. Bhattacharyya

Department of Electrical and Computer Engineering, University of Maryland
College Park, MD, 20742, USA, Email: rgu, ssb@umd.edu
Xilinx, Inc. San Jose, CA, USA, Email: jorn.janneck@xilinx.com
IETR Laboratory, UMR CNRS 6164, Image and Remote Sensing Group,
35043 RENNES Cedex, FRANCE, Email: mraulet@insa-rennes.fr

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 [1], have been developed to describe dynamically-structured DSP applications. Such generalized models can express dynamically changing functionality, but lose the powerful static scheduling capabilities provided by SDF. This paper focuses on detection of SDF-like regions in dynamic dataflow descriptions — in particular, in the generalized specification framework of CAL. This is an important step for applying static scheduling techniques within a dynamic dataflow framework. Our techniques combine the advantages of different dataflow languages and tools, including CAL [1], DIF [2] and CAL2C [3]. The techniques are demonstrated on the IDCT module of MPEG Reconfigurable Video Coding (RVC).

Index Terms— CAL, dataflow, quasi-static scheduling.

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 utilize statically scheduling techniques, as in SDF.

In the context of DSP system design, dataflow programs consist of computational kernels, called *actors*. Actors are connected to each other by FIFO channels, called *edges*, through which they send each other packets of data, called *tokens*. Actors execute iteratively through discrete units of execution called *firings* or *invocations*. 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 variety of dataflow-based languages and tools have been developed. 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 semantics that underlies the CAL language bears some similarity to the stream-based functions model of computation [6]. 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] 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.

In this paper, we explore an integration of CAL, TDP, and CAL2C to provide 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.

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 CAL language. The efficiency of quasi-static schedules built from SSRs is demonstrated by evaluating synthesized C-code implementations that are generated using CAL2C.

2. ANALYSIS FRAMEWORK

Our method to optimize implementation of DSP applications combines the advantages of three complementary tools, as shown in Figure 1. The given DSP application is initially described as a CAL network (a highly expressive form of dataflow graph) 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 structured provided by the SSRs and their enclosing quasi-static schedule.

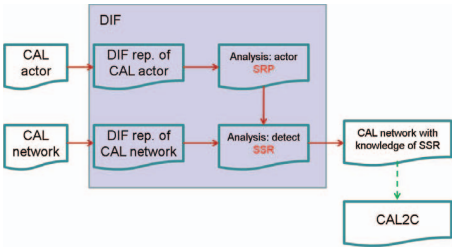

Fig. 1. Outline of method for optimizing dataflow graph implementation.

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

We introduce some notation to allow for more detailed discussion of CAL semantics. For simplicity, we assume here that there is exactly one state variable associated with a given CAL actor, but this is not a general restriction of the CAL language — CAL actors can have no state variables or multiple state variables. A CAL actor A can be represented as a 4-tuple $< \sigma_0, \Sigma(A), \Gamma(A), \succ >$, where $\Sigma(A)$ is the set of all possible values for the state variable; $\sigma_0 \in \Sigma(A)$ is the initial state; $\Gamma(A)$ is the set of all possible actions for actor A; and $\succ$ is a non-reflexive, anti-symmetric and transitive partial order relation on $\Gamma(A)$ called the *priority relation* of A. Intuitively, if $l, m \in \Gamma(A)$, then $l \succ m$ means that l has priority over m if both are "competing" for the next invocation A.

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

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 formed from all ordered pairs of ports $(A.a, B.b)$ such that there is an edge in G representing a connection from port $A.a$ to port $B.b$. When discussing a graphical representation of a CAL network, we assume that the representation is in the form of a PCG, unless otherwise stated.

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

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

3. STATICALLY SCHEDULABLE REGIONS

As a core step of our proposed design flow, we identify certain groups of related ports in the PCG that is derived from a given CAL program. Intuitively, these groups are to be treated as single units during subsequent stages of analysis, similar to how graph clustering techniques are used to group sections of

a graph for isolated analysis. Our particular method of grouping is driven by the goal of constructing efficient quasi-static schedules. Our grouping process operates in two phases: in phase 1, we group ports within individual actors, and in phase 2, we group ports across distinct actors.

In phase 1, we apply a concept that we refer to as *coupled ports*, which is based on the observation that within a given actor A, two ports a and b can be related to one another in zero, one or both of the following two ways:

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.

If ports a and b satisfy one or both of these relationships, we say that the two ports are *coupled* to one other.

Our process of grouping ports in a PCG starts by examining the PCG subgraph associated with each actor A, and building a set of coupled regions of ports within the subgraph. This grouping process follows a simple iterative scheme where a given group g is incrementally extended by testing all ports in A that have not yet been grouped (external ports) to find an external port that is coupled to at least one port inside g. By following this scheme, we eventually arrive at a partition of the ports within an actor into a set of one or more *coupled groups*.

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. For each input port $p_i \in Z$, 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, for each output port $p_j \in Z$, 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.

In general, when applying SRG detection to a coupled group c containing x ports, we will arrive at an SRG with y ports, where $0 \leq y \leq x$. The remaining $(x - y)$ ports in c (i.e., the ports that lie outside the SRG), are referred to as *dynamic ports* of c.

In phase 2 of our overall port-grouping process, we cluster *connected* SRGs across distinct actors to form statically schedulable regions (SSRs). We define connectedness for SRGs as follows. Suppose that G is a PCG, A and B are distinct actors in G, Z_a is a non-empty SRG of A, and Z_b is a non-empty SRG of B. Then Z_a and Z_b are connected if there exists an edge (p_a, p_b) in the PCG G such that $p_a \in Z_a$ and $p_b \in Z_b$.

Intuitively, two SRGs are connected if there is at least one PCG edge that connects ports across the two SRGs.

In phase 2 of our port-grouping process, we incrementally cluster *connected subsets* of SRGs by iteratively adding a new SRG to an existing connected subset S whenever Z_n from outside of S is found such that Z_n is connected to at least one SRG in S. Using this kind of process, a unique set of *maximal connected subsets* of SRGs emerges; such a maximal connected subset is precisely what we mean by an SSR.

A practical example of grouping into SSRs is shown in Figure 2. This example is part of a variable polyphase video scaler that has been modeled fully in CAL. The shaded regions shown in the figure correspond to the different SSRs, which are unique to the application, and detected systematically using the two-phase, port-grouping process that we have outlined above.

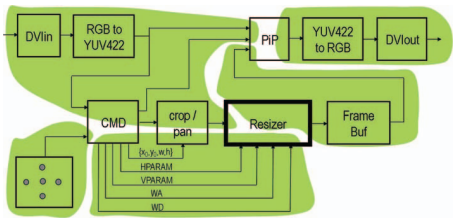

Fig. 2. Example of SSR regions in multi-media processing.

4. CASE STUDY: IDCT

The MPEG4 RVC framework aims at providing a new, interoperable model of defining MPEG standards at the system-level [7]. C code for an MPEG RVC decoder can be generated automatically in CAL2C. However, CAL2C utilizes scheduling mechanisms that are embedded in SystemC, which are not optimized in terms of static or quasi-static scheduling.

The IDCT is one key element in the block diagram of an RVC decoder. In the original C code generated by CAL2C, each actor has an *action scheduler*, which schedules the actions through a control structure that is similar to a finite state machine. The action to be executed in the next step (invocation) is determined in real-time by outputs from the previous step, and the current input to the actor.

In our experiments, we first translated the CAL network for the IDCT subsystem into a DIF-based intermediate representation in TDP. This translation was performed through a conversion tool that we have developed for automatically translating from CAL to TDP. We then applied our algorithm

for SSR detection, which we have implemented within TDP. The output of SSR detection, along with a static schedule for each detected SSR, is then translated back to CAL, thereby achieving a source-to-source transformation of the original CAL network through TDP.

Figure 3 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. There are two separate shaded regions in the graph, representing two distinct SSRs.

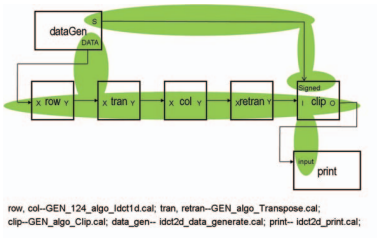

Fig. 3. SSRs in the IDCT subsystem.

We generated C code 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. The second version (V2) exploits the SSRs illustrated in Figure 3, and employs a quasi-static integration of static schedules for these SSRs with top-level dynamic scheduling. The third version (V3) uses a modified, more predictable version of the *clip* actor that can be used when the input data is known in advance. In V3, the entire IDCT is subsumed by a single SSR, and therefore, purely static scheduling can be achieved.

We experimented with all three IDCT versions using Microsoft Visual Studio on a general purpose computer. While these results provide a useful form of comparison — e.g., in the context of dataflow-based system simulation — it would be interesting to perform analogous experiments on a programmable digital signal processor platform. This is a useful direction for further study.

Our experimental results are shown in Figure 4. Here, V2 shows an improvement in performance of 1.5 times compared to V1, whereas V3 shows the best performance among all three versions.

5. 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.

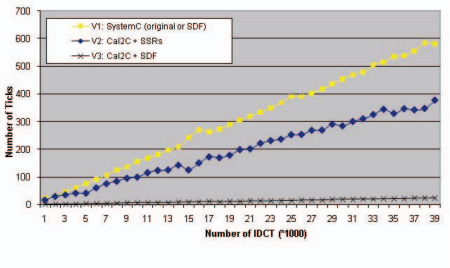

Fig. 4. Results: clock cycles vs number of iterations.

6. REFERENCES

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

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

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

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

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

[7] C. Lucarz, M. Mattavelli, J. Thomas-Kerr, and J. Janneck, "Reconfigurable media coding: A new specification model for multimedia coders," in *Proceedings of IEEE Workshop on Signal Processing Systems*, October 2007, pp. 481–486.