Classification and transformation of dynamic dataflow programs

Matthieu Wipliez, Mickael Raulet

To cite this version:

Matthieu Wipliez, Mickael Raulet. Classification and transformation of dynamic dataflow programs. Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on, 2010, United Kingdom. pp.303 -310, 2010, <10.1109/DASIP.2010.5706280>. <hal-00565290>

HAL Id: hal-00565290

https://hal.archives-ouvertes.fr/hal-00565290

Submitted on 11 Feb 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

CLASSIFICATION AND TRANSFORMATION OF DYNAMIC DATAFLOW PROGRAMS

Matthieu Wipliez, Mickaël Raulet

IETR/INSA
UMR CNRS 6164
F-35043 Rennes, France
Email: mwipliez@insa-rennes.fr
Email: mraulet@insa-rennes.fr

ABSTRACT

Dataflow programming has been used to describe signal processing applications for many years, traditionally with cyclo-static dataflow (CSDF) or synchronous dataflow (SDF) models that restrict expressive power in favor of compile-time analysis and predictability. Dynamic dataflow is not restricted with respect to expressive power, but it does require runtime scheduling in the general case. Fortunately, most signal processing applications are far from being entirely dynamic, and parts with static behavior need not be dynamically scheduled. This paper presents a method to automatically analyze and classify blocks of a dynamic dataflow program within more restrictive dataflow models when possible, and to transform the blocks classified as static to improve execution speed by reducing the number of FIFO accesses. We used this method on actors of two dynamic dataflow descriptions of an MPEG-4 part 2 decoder, and study how classification and transformation increases decoding speed.

Index Terms— Dataflow Programming, Classification, RVC-CAL, Abstract Interpretation

1. INTRODUCTION

The arrival of multi-core in the desktop computing market has renewed the interest in multi-processor programming. There are many different techniques to write multi-processing programs, depending on memory architecture (shared or distributed), architecture (uniprocessor or multiprocessor), number of cores (single core, multi-core, many-core), etc. Most of these techniques constrain program design in a way that make it difficult to use a different technique, should the program be ported to a different architecture than initially planned. Dataflow programming is a portable platform-agnostic alternative that allows an algorithm to be described so that parallelism is made explicit.

A dataflow description consists in a directed graph where vertices (or *actors*) process data and edges carry data, with the requirement that vertices cannot *share* data. Actors can only communicate with other actors through ports connected to edges. The semantics of a dataflow program are defined by a Model of Computation (MoC) that dictates conditions for existence of a valid schedule, bounded memory consumption, proof of termination, and other properties. MoCs go from Synchronous Dataflow (SDF) with total compile-time predictability with respect to scheduling, memory consumption, termination, to dynamic dataflow where those properties are not predictable in the general case, with increasing expressiveness, for instance see [1–6].

The Reconfigurable Video Coding (RVC) [7] standard defines existing MPEG video standards as dynamic dataflow programs in which actors are written in a language called RVC-CAL. These dataflow programs can be automatically translated and ported to a wide range of platforms and languages, from hardware to multi-core software [7]. Contrary to SDF and a few other MoCs, RVC-CAL has a much greater expressive power, but this also means that runtime scheduling is mandatory in the *general* case, which severely impacts execution speed.

Fortunately, most signal processing applications are far from being entirely dynamic, and parts with static behavior need not be dynamically scheduled. The problem is to detect actors that behave statically or quasi-statically, since dynamic dataflow has an expressive power equivalent to a Turing machine [4], which means it is not possible to prove the termination of a dynamic dataflow program in general. This paper makes the following contributions:

- we present a method to automatically classify an actor as *static*, *cyclo-static*, or *quasi-static* (section 3). As shown on Fig. 1, classification is the first step towards high-performance implementation of dataflow programs. It is mandatory to allow other optimizations to take place.

- we describe an algorithm to transform actors classified as “not dynamic” to reduce scheduling overhead when executing those actors (section 4). The transformation we describe may be used to transform actors into other actors that will not only execute faster, but will also facilitate the application of later optimizations. Indeed, actor merging consists of creating *composite* actors from several actors, and our transformation changes actors so that they can be merged more easily. The last optimization that can be used to achieve the maximum throughput for a dataflow application is multiprocessor mapping and scheduling techniques for statically schedulable subsets (for an example see [8–10]).

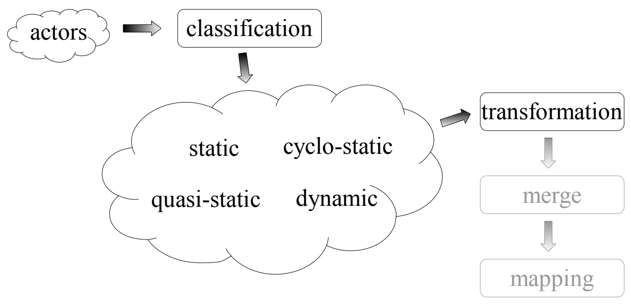

Fig. 1. Towards an efficient implementation of dynamic dataflow programs using classification and transformation.

2. BACKGROUND

2.1. Dataflow Programming with RVC-CAL

This section presents the model of the programs that we classify and transform. Formally, the dataflow programs we are dealing with are *Dataflow Process Networks* (DPNs) [6]. DPNs are related to Kahn Process Networks (KPNs) [11] as both models contain blocks (processes in a KPN, actors in a DPN) that communicate with each other through unidirectional, unlimited FIFO channels. Reading from a FIFO is *blocking* (if a process or an actor attempts to read data from a FIFO and no data is available, it must wait), whereas writing to a FIFO is *non-blocking*, i.e. the write returns immediately. Programs that respect one model or the other must be scheduled dynamically [6, 12]. Contrary to Kahn processes, actors can test an input port for the absence or presence of data ; an actor need not be suspended when it cannot read, which in turn means that scheduling a DPN does not require context-switching nor concurrent processes.

MPEG has recently defined a standard called Reconfigurable Video Coding (RVC) [7] to describe existing and future video standards with dataflow programming. A dataflow program consists of a hierarchical DPN where actors are defined with the RVC-CAL language, a Domain-Specific Language (DSL) for signal processing. An RVC-CAL actor may have input and output ports connected to FIFOs, parameters, state variables, functions and procedures, *actions* that may be identified by a tag, a Finite State Machine (FSM) whose transitions reference tagged actions, and a set of priorities that establish a partial order between action tags.

Contrary to most non-dataflow languages, dataflow programs do not admit a single entry point. A dataflow network has one entry point per actor, which means the program may start from any actor in the network. The only entry points of an actor are its actions; functions and procedures can only be called by an action. An action computes data by reading tokens from input ports and writing tokens to output ports. The patterns of tokens read and written by a single action are called input pattern and output pattern respectively. Additionally an action has *guards* that must be true for an action to be executed. Apart from these specific features, the body of an action is like a procedure in most imperative programming languages, with local variables and imperative statements.

An actor is executed (or *fired*) by selecting a *fireable* action and firing it. An action is fireable iff: (1) the current FSM state allows the action to fire (or there is no FSM and this condition is always true), (2) there are enough tokens for the action to fire, (3) the guards of the action evaluate to `true`.

```
actor Clip ()
  int(size=10) I, bool SIGNED ==> int(size=9) O :

  int(size=7) count := -1;
  bool sflag;

  read_signed: action SIGNED:[s] ==>
  guard count < 0
  do
    sflag := s;
    count := 63;
  end

  limit: action I:[i] ==> O:[ if i > 255 then 255
    else if i < min then min else i end
    end ]
  var
    int min = if sflag then -255 else 0 end
  do
    count := count - 1;
  end

  priority
    read_signed > limit;
  end

end
```

Fig. 2. The `Clip` actor in RVC-CAL.

Figure 2 shows an actor named “Clip” that performs a *clipping* operation. This operation is traditionally done in video decoders after an inverse Discrete Cosinus Transform to clip the values of pixels to a given interval. The actor has the following behavior. When a boolean token arrives on the *SIGNED* port, the **read_signed** action fires, reads the token and stores it in the `sflag` state variable, and initializes a counter named `count`. The `sflag` variable configures the clip as signed if `true` and unsigned otherwise. At this point only the **limit** action can fire, regardless of whether there are tokens on the *SIGNED* port, because `count` is nonnegative. Each time this action fires, it reads a token on *I*, decrements the counter, clips the value read to $[-255, 255]$ in signed mode or to $[0, 255]$ in unsigned mode, and outputs the clipped value on *O*. When `count` reaches -1, the cycle can start again.

2.2. Classification of an RVC-CAL Actor: An Example

Figure 3 shows the behavior of the "Clip" actor expressed with Cyclo-Static Dataflow (CSDF) [2]. Within this model, the number of tokens produced and consumed by an actor changes periodically according to a production/consumption sequence. For instance, the consumption sequence on *IN* has a zero followed by 64 ones: On its first invocation, the actor will not consume any token on the port, and for the next 64 invocations, it will consume one token on the port each time. The advantage of expressing an actor as CSDF is that it can be statically scheduled because the number of tokens the actor will consume and produce is known at compile-time.

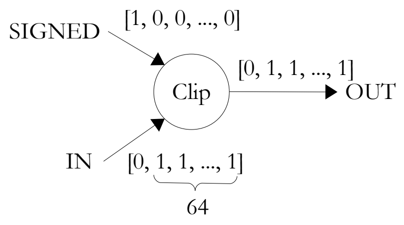

Fig. 3. Behavior of the "Clip" actor.

The problem is then how to classify actors into Models of Computation (MoCs) more constrained than dynamic dataflow so that as many actors as possible can be statically or quasi-statically scheduled.

2.3. A Taxonomy of Dataflow Models of Computation

This section presents a taxonomy of Models of Computation (MoCs) (Fig. 4) that can model the different types of behavior that actors can exhibit. The actors in the Video Tool Library of the RVC standard can behave statically, cyclo-statically, quasi-statically, or dynamically. We present the MoCs in the literature that are suitable to model these types of behaviors.

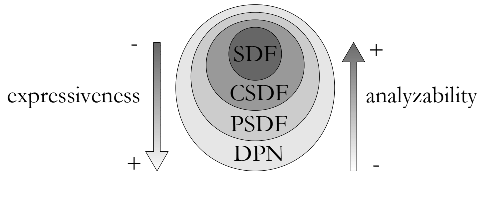

Fig. 4. Dataflow Models of Computation.

Dataflow MoCs are defined as subsets of a general model called Dataflow Process Networks (DPNs). The taxonomy shown on Fig. 4 reflects the fact that MoCs are progressively restricted from DPN towards SDF with respect to expressiveness, but at the same time they become more amenable to analysis. We first study the rules of DPN, and then present the models that can be used to model static, cyclo-static, quasi-static, and dynamic RVC-CAL actors.

Dataflow models respect the semantics of DPNs: A dataflow model is a directed graph whose vertices are *actors* and edges are unidirectional FIFO channels with unbounded capacity, connected between ports of actors. Each FIFO channel carries a sequence of tokens. Executing a DPN boils down to executing repeatedly the actors in the graph, possibly *ad infinitum*. An actor executes (or *fires*) when at least one of its *firing rules* is satisfied. Each firing may consume and produce tokens. An actor can have N firing rules, where each one represents an acceptable sequence of tokens. Additionally an actor has a firing function that takes a sequence of tokens and produce a sequence of tokens.

Synchronous Dataflow (SDF) [1] is the least expressive DPN model, but it is also the model that can be analyzed more easily. Schedulability and memory consumption of SDF graphs can be determined at compile-time, and algorithms exist that can map and schedule SDF graphs onto multiprocessors in linear time with respect to the number of vertices and processors [8]. Any two firing rules of an SDF actor must consume the same amount of tokens, and all firings must produce the same amount of tokens on the output ports. This definition is actually included in Lee's denotational semantics for SDF [6], which states that SDF actors have a single firing rule. Our definition simply allows SDF actors to have several firing rules as long as they have the same production/consumption rate, which in practice makes it easier to describe SDF actors that have data-dependent computations.

Cyclo-static Dataflow (CSDF) [2] extends SDF with the notion of *state* while retaining the same compile-time properties concerning scheduling and memory consumption. State can be represented as an **additional argument** to the firing rules and firing function, in other words it is modeled as a self-loop. Contrary to the RVC-CAL version of "Clip", the state of the CSDF version does not contain the `sflag` variable because it is not taken into account in firing rules, instead it only consists of the value of the `count` variable that holds the number of values processed so far.

Synchronous and cyclo-static dataflow allow signal processing algorithms to be modeled as graphs with fixed production/consumption rates. On the other hand, so-called "quasi-static" graphs can be used to describe data-dependent token production and consumption. Quasi-static dataflow differs from dynamic dataflow in that there are techniques that statically schedule as many operations as possible so that only data-dependent operations are scheduled at runtime [3, 4, 9]. We chose to use the PSDF [3] model as a target for our classification because it can be used to model static, cyclo-static and quasi-static behavior as a dataflow graph.

RVC-CAL places no restrictions whatsoever about the firing rules nor firing function of an actor. An RVC-CAL actor can thus have a behavior that is data-independent and state-independent, cyclo-static state-dependent, quasi-static data-dependent, or data-dependent and state-dependent (dynamic). The RVC-CAL language extends the DPN MoC by adding a notion of **guard** to firing rules. Formally the guards of a firing rule are boolean predicates that may depend on the input patterns, the actor state, or both, and must be `true` for a firing rule to be satisfied.

3. AUTOMATIC CLASSIFICATION OF DYNAMIC DATAFLOW ACTORS

This section presents the first contribution of this paper: A method to automatically classify dynamic dataflow actors into more restricted dataflow Models of Computations. Classification is a prerequisite for the transformations presented in section 4 and actor merging. We first present how the classification method can detect actors that it cannot classify and discard them.

3.1. Detection of Unclassifiable Actors

DPN places no restrictions on the description of actors, and as such it is possible to describe a **time-dependent** actor in that its behavior depends on the time at which tokens are available. This happens in RVC-CAL when a given action reads tokens from input ports not read by a higher-priority action, and these actions have guards that are not mutually exclusive.

The "Clip" actor presented section 2 has a time-dependent behavior. If you read the description carefully, you will notice that the **limit** action can fire as long as there tokens on *IN* and not on *SIGNED*, therefore potentially clipping pixels *before* clipping mode is set by **read_signed**. In this particular case, this behavior is a flaw in the implementation of the actor itself, although it may never cause any problems if the actors connected to it always send a token on the *SIGNED* port first. In other cases, time-dependent behavior can be useful as a low-level optimization by allowing an actor to test for the absence of data and still do something useful when that is the case. Time-dependent behavior can be removed in some cases simply by making guards mutually-exclusive, which in our example translates to rewriting the **limit** action as presented in Fig. 5.

Classifying a time-dependent actor may be intractable or impossible depending on the kind of actor. This kind of actor cannot be classified as SDF by definition (because actions of an SDF actor have the same token production/consumption rate), but it *could* still be considered a valid cyclo-static or quasi-static actor, in which case we would need to record the sequences of tokens that lead to this cyclo-static or quasi-static behavior. The intractability of classifying such an actor lies just there, since an automatic classification algorithm

```
limit: action I:[i] ==> O:[ if i > 255 then 255
  else if i < min then min else i end
  end ]
var
  int min = if sflag then -255 else 0 end
guard count >= 0 // mutually exclusive with read_signed
do
  count := count - 1;
end
```

Fig. 5. The **limit** action rewritten in a time-independent way.

would need to explore all the possible input patterns with no criterion as to when to stop the enumeration of possible tokens.

Therefore our classification method must detect and discard time-dependent actors to prevent enumerating the universe of possible token sequences. Detecting actions that read input ports not read by higher-priority actions is trivial. However, such actions may not render the actor time-dependent if their guards are exclusive, which must be mechanically verified. To this end we feed the guards to a constraint solving system, which either gives the values of tokens and state variables that satisfy both guards (guards not mutually exclusive), or else finds no solution (guards mutually exclusive).

Constraints are created from the guards of an action as follows. The guards of an action are boolean expressions that must be simultaneously true for the action to be fired, so we translate each guard of any two actions to a constraint. Suppose an action has a guard $x > 0$ and the other action has a guard $x \neq 1$, then the constraint solver will find values of x that satisfy the constraints, such as $x = 2$. If there is such a solution, this means the guards are not mutually exclusive.

3.2. Abstract Interpretation of Actors

Classifying an actor within a MoC is based on checking that a certain number of MoC-dependent rules hold true for any execution of this actor. Some of these rules are verified solely from the structural information of the actor, for instance the rules for a static actor only depends on the input and output patterns of actions. In more complicated cases, we need to be able to obtain information from an actual execution. The actor must be executed so that the information obtained is valid for *any* execution of the actor, whatever its environment (the values of the tokens and the manner in which they are available). As a consequence it is not possible to simply execute the actor with a particular environment supplied by the programmer. To circumvent this problem we use *abstract interpretation* [13].

Abstract interpretation consists in doing the computations performed by a program in an abstract universe of objects rather than on concrete objects. Our abstract interpretation of an actor has the following properties:

- The set of values that can be assigned to a variable is

$$Values = \mathbb{Z} \cup \{\texttt{true}, \texttt{false}\} \cup \{\bot\}$$

The value $\perp$ is used for variables whose value is unknown, e.g. for uninitialized variables.

- The environment is defined as an association of variables and their values:

$$Env : Idents \rightarrow Values$$

Env initially contains the state variables of the actor associated with their initial value if they have one, otherwise with $\perp$.

- When the interpreter enters an action, the environment is augmented with bindings between the name of the tokens in the input pattern and $\perp$. In other words, a token read has an unknown value by default.

The abstract interpreter interprets an actor by firing it repeatedly until either one of the conditions is met:

1. The interpreter is told to stop because analysis is complete as determined by the classification algorithm.
2. The interpreter cannot compute if an action may be fired because this information depends on a variable whose value is $\perp$.

To fire the actor, the interpreter starts by selecting one fireable action, that is an action that meets the criteria defined section 2.1. The abstract interpretation of an RVC-CAL actor is the same as its concrete interpretation with the following exceptions. Any expression that references a variable v where $Env(v) = \perp$ has the value $\perp$. Conditional statements and loops that test an expression whose value is $\perp$ are not executed. However, guards evaluated as $\perp$ cause the abstract interpreter to stop as per condition 2.

3.3. Classification of a static actor

Classification tries to classify each actor within models that are increasingly expressive and complex. The rationale behind this is that the more powerful a model, the more difficult it is to analyze. If an actor cannot be classified as a static actor, the method will try to classify it as cyclo-static, and then as quasi-static. An actor is classified as *static* iff it conforms to the SDF MoC, which means that all its actions have the same input and output patterns. A one-action actor is by definition static.

3.4. Classification of a cyclo-static actor

The conditions an actor must meet to be a candidate for classification as cyclo-static are two-fold: (1) it must have a *state*, hereinafter noted $\mathcal{S}$, and (2) there must be a fixed number of data-independent firings that depart from the initial state, modify the state, and return the actor to its original state $\mathcal{S}_0$. We consider two kinds of actor state:

1. $\mathcal{S}$ consists of a set of scalar state variables and their runtime value. A state variable belongs to $\mathcal{S}$ iff it has an initial value and is used in at least one guard expression. $\mathcal{S}_0$ is the set of variables of $\mathcal{S}$ with their initial value. Non-scalar variables (arrays) are not taken into account because state is typically not implemented with them. A full *cycle* is found when at least one action has been executed, and $\mathcal{S} = \mathcal{S}_0$ is true.
2. In the case where $\mathcal{S} = \emptyset$ and the actor does not have an FSM, it is considered to have no state and therefore cannot be classified as cyclo-static. Otherwise the state consists of the current FSM state, and $\mathcal{S}_0$ is the initial state s_0 of the FSM: $\mathcal{S}_0 = s_0$. If there is no path that returns $\mathcal{S}$ to $\mathcal{S}_0$, the actor cannot be classified as cyclo-static.

Once the classification algorithm finds the actor to be a valid cyclo-static candidate, we use the abstract interpreter presented section 3.2 until we find that the actor has returned to its original state, or the abstract interpreter stops because of data-dependent behavior. When the actor has returned to its original state, the algorithm stores the sequence of actions that fired, as well as the production and consumption of tokens on each port of the actor.

3.5. Classification of a quasi-static actor

A quasi-static actor is informally described as an actor that may exhibit distinct static behaviors depending on a data-dependent condition. Our classification method is restricted to classify the subset of quasi-static actors considered by Boutellier et al. [9] and defined as follows. A quasi-static must have an FSM whose initial state s_0 has transitions to n *branches* ($n \geq 2$), the i^{th} branch starting with state s_i.

Each transition from s_0 to s_i must be solely dependent on a *control token*; s_0 may have a cycle, which simply consumes one control token and returns the actor to the initial FSM state. Self-loops and cycles more generally are allowed within a branch, and so are cross-branch transitions, as long as all branches go back to the initial state.

The first step of the classification of an actor as quasi-static is to assert it has an FSM that respects the aforementioned conditions. This is done simply by examining each successor s_i of the initial state s_0, and checking that there is a path from s_i back to s_0. This criterion alone does not qualify the actor as quasi-static, it merely discards candidates that cannot be quasi-static.

The second step of the classification consists in checking that each branch fires a fixed number of data-independent firings and returns to the initial state:

1. for each branch i, find a value that satisfies the condition to take branch i but not any branch before it. We use constraint solving to automatically find a satisfying value.

2. use the abstract interpreter by taking branch i and firing the actor until it goes back to the initial state, or the abstract interpreter stops because of data-dependent behavior.

Taking branch i is done by making the interpreter return the concrete value computed in step 1 instead of $\perp$ when the control port is read. It is important that the abstract interpreter only provide the concrete value **once**. Indeed, in some FSMs there may be more than one conditional state, i.e. more than one state being conditioned by the control port. We could probably further narrow the subset of acceptable actors with this criterion, but this is not necessary since the abstract interpreter will identify the transitions departing from a conditional state different from s_0 as data-dependent.

4. TRANSFORMATION OF CLASSIFIED ACTORS

This section presents the second contribution of this paper: A method to automatically transform actors that were classified as static, cyclo-static, or quasi-static, to higher-level SDF and PSDF graphs. This transformation improves execution speed of the resulting actors, and makes merging actors of the same kind easier.

4.1. Transformation to SDF and PSDF

The classification of actors gives information about the sequence of actions that were fired:

- in the case of static behavior, there may only be one action by definition; actors that have several actions with similar input/output patterns must be transformed to single-action actors.
- in the case of cyclo-static behavior, the sequence is a list of actions with fixed production/consumption rates that start from an initial state and eventually return to this initial state.
- in the case of quasi-static behavior, there are several sequences of actions; each sequence is a concatenation of a first conditional action and a sequence of actions with fixed production/consumption rates.

To allow actors to be merged later, these sequences must be transformed to respect appropriate MoCs. They can be trivially transformed from cyclo-static to CSDF and from quasi-static to PSDF, by transforming an action invocation to a vertex and setting production and consumption to zero on every edge (since the actions do not consume the data of one another). However, this kind of graphs cannot be easily manipulated or optimized. The graphs do not represent the behavior of the actors; they are not suitable for merging, in particular merging SDF graphs together when each graph is composed of up to a few hundred vertices can quickly result in huge graphs, especially if the repetitions of vertices are not multiple of one another (see [1] for additional explanations); the graphs cannot be efficiently mapped and scheduled because optimally scheduling a SDF graph is an NP-complete problem [8].

A better representation is a graph where a sequence of actions is transformed to a single higher-level action that fires all the actions in the sequence consecutively, this way edges can carry the proper production/consumption rates and the graph accurately represents the actor's behavior. The contents of higher-level actions can be factorized with loop *rerolling*.

Loop rerolling is the exact opposite of the well-known loop unrolling transformation. It has been used by Stitt and Vahid to recover loop structures from compiled code [14]. In the context of this work, we used this transformation to find loops of actions within an initially flat sequence of actions. Loops are rerolled in two steps. First, we must recognize common sequences within an input sequence. Then, loops can be formed around consecutive repetitions of common sequences.

We used the Sequitur [15] algorithm to recognize common sequences of actions from the initially flat list of actions. Sequitur works by deriving a hierarchical structure in the form of a Context-Free Grammar from a sequence of symbols. For instance the grammar G_1 derived from the sequence of symbols `ababc` is: S $\rightarrow$ A A c, A $\rightarrow$ a b.

To obtain loops from the hierarchical structure, we walk through the hierarchical structure by counting the number of rule invocations and developing the rules. For instance, suppose we have a sequence[1] composed of four repetitions of a sub-sequence composed of five `a`s followed by three `b`s, noted 4(5(a) 3(b)). The corresponding grammar is transformed so that we group consecutive actions together, which gives us 5(a) 3(3(b) 5(a)) 3(b). Sequitur works in linear time, so the hierarchical structure it derives is not optimal, which explains why the result is not minimal in terms of number of loops.

4.2. Optimization of Action Scheduler

Before classification, all actors are considered dynamic. This means that to fire an action an **action scheduler** must check that there are enough tokens in the input FIFOs and enough room in the output FIFOs, read tokens, compute data, and write tokens. After classification we know that some actors have a behavior that is static, cyclo-static, or quasi-static. As a consequence, we have information about the number of tokens and the room needed for *several* actions to fire, not just one.

A static actor is transformed to an actor with a single action, so it is not possible to reduce the number of read and write operations. Conversely, after loop rerolling cyclo-static actors have one high-level action, and quasi-static actors have n conditioned high-level actions. Those high-level actions act

[1] The developed sequence is "aaaaabbbaaaaabbbaaaaabbbaaaaabbb".

as *static* action schedulers: They fire sequences of actions, each action potentially reading and writing tokens. Since we know the number of firings that will occur, those reads and writes can be replaced by loads from/stores to arrays. For instance the **limit** action of "Clip" would be transformed as shown on Fig. 6. The **A** action is transformed as follows:

1. read data from each input *port* in a `tokens_`*`port`* array
2. initialize each `index_`*`port`* variable to zero
3. fire actions
4. write data to each output *port* from `tokens_`*`port`*

```
limit: action ==>
var
  int(size=10) i := tokens_I[index_I],
  int(size=9) o,
  int min = if sflag then -255 else 0 end
do
  index_I := index_I + 1;
  count := count - 1;
  o := if i > 255 then 255
    else if i < min then min else i end
    end;
  tokens_O[index_O] := o;
  index_O := index_O + 1;
end
```

Fig. 6. The **limit** action transformed.

5. RESULTS

We have implemented the classification method and the transformation algorithm in the RVC-CAL compilation infrastructure supported by the Open RVC-CAL Compiler (Orcc)[2]. Orcc has a front-end that compiles RVC-CAL actors to an Intermediate Representation (IR) that retains structural information but lowers the RVC-CAL language to a simpler language in SSA form [16]. The IR of actors can then be loaded and transformed to source code in any of the target languages supported by back-ends, namely C, C++, Java, LLVM, and VHDL.

We used the Cream [17] constraint programming library to check mutually exclusive guards and to find values in the quasi-static classification method. Cream is capable of solving constraints defined on integers using different branch-and-bound strategies. We extended the library to handle bitwise `and` constraints.

The classification method has been tested on 50 actors used by two dataflow descriptions of an MPEG-4 part 2 decoder present in Orcc. Table 1 shows the classification results with actors classified as static, cyclo-static, quasi-static, dynamic, time-dependent.

[2]Orcc is available at: `http://orcc.sf.net`

Number of actors	Classification
6	static
14	cyclo-static
11	quasi-static
13	dynamic
6	time-dependent

Table 1. Classification results on 50 actors.

The action scheduler optimization has been implemented for cyclo-static actors with mono-token reads and writes. This subset allowed us to test only one of the two dataflow descriptions, in which it resulted in a 20% increase of the number of frames decoded per second. Measurements on a particular actor responsible of the interpolation in motion compensation indicate that the transformed version of the actor is 2.4 times faster than the original version.

6. RELATED WORK

Zebelein et al. present a classification algorithm for dynamic dataflow models in [18]. In their model, actors are defined as SystemC modules that receive and send data via SystemC FIFOs. Their classification method is based on the analysis of read and write patterns and FSMs of the different modules. Compared to ours, their approach is limited by the fact that they ignore any C++ code that does not contain a read or a write, and that they do not classify quasi-static actors.

In [9], Boutellier et al. show how to express quasi-static RVC-CAL actors as PSDF graphs and how to derive a multiprocessor schedule from these graphs. However, they do not address the issues of automated classification and transformation: Quasi-static behavior is specified with parameters defined *manually*, and they do not explain how low-level Homogeneous SDF (HSDF) graphs created from quasi-static branches can be automatically transformed to high-level PSDF graphs. As a consequence, we believe that our work can serve as a preprocessing step for their approach by automatically classifying actors as quasi-static and transforming them to high-level PSDF graphs.

Gu et al. present a technique to recognize a set of Statically Schedulable Regions (SSRs) within a dynamic dataflow program [10]. SSRs are sets of ports that are *statically coupled*, which essentially means that the production of an output port matches the consumption of the input port(s) it is connected to (additional criteria are developed in [10]). On the one hand, SSR classification has potentially more knowledge about static behavior because it looks at connected actors rather than just inside actors. On the other hand, by considering an actor as a whole our classification can discover its behavior (cyclo-static and quasi-static) and transform it into a high-level SDF or PSDF graph that will make merging easier. Using SSRs to obtain additional information as an input to our classification algorithm is a possible direction for future work.

7. CONCLUSION

This paper presented a method to automatically classify dynamic dataflow actors into more restricted dataflow MoCs, along with a method to automatically transform classified actors to static dataflow and parameterized static dataflow graphs. The transformations presented allow more efficient code to be generated for those actors and improve execution speed by reducing the number of FIFO accesses. We used these methods on 50 actors from two dynamic dataflow descriptions of an MPEG-4 part 2 decoder, and studied how the actors that could be transformed following classification could improve execution speed by 20% on one of the descriptions.

The work described in this paper paves the way for future work concerning the merge of SDF and PSDF actors together. Merging actors would further reduce the overhead induced by FIFO accesses, and would also reduce the amount of dynamic scheduling currently needed. Actor merging is a necessary step towards efficient multi-processor mapping and scheduling by increasing the computation to communication ratio.

8. REFERENCES

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

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

[3] Bishnupriya Bhattacharya and Shuvra S. Bhattacharyya, "Parameterized Dataflow Modeling for DSP Systems," *IEEE Transactions on Signal Processing*, vol. 49, pp. 2408–2421, 2001.

[4] J.T. Buck and E.A. Lee, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," *Acoustics, Speech, and Signal Processing, IEEE International Conference on*, vol. 1, pp. 429–432, 1993.

[5] J.T. Buck, "Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams," in *Presented at 28th Asilomar Conference on Signals*. Citeseer, 1994.

[6] Edward A. Lee and Thomas M. Parks, "Dataflow Process Networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.

[7] M. Mattavelli, I. Amer, and M. Raulet, "The Reconfigurable Video Coding Standard [Standards in a Nutshell]," *Signal Processing Magazine, IEEE*, vol. 27, no. 3, pp. 159 –167, may 2010.

[8] M. Pelcat, J. Piat, M. Wipliez, S. Aridhi, and J.F. Nezan, "An open framework for rapid prototyping of signal processing applications," *EURASIP Journal on Embedded Systems*, vol. 2009, pp. 3, 2009.

[9] J. Boutellier, C. Lucarz, S. Lafond, V.M. Gomez, and M. Mattavelli, "Quasi-static scheduling of CAL actor networks for reconfigurable video coding," *Journal of Signal Processing Systems*, pp. 1–12, 2009.

[10] R. Gu, J.W. Janneck, S.S. Bhattacharyya, M. Raulet, M. Wipliez, and W. Plishker, "Exploring the concurrency of an MPEG RVC decoder based on dataflow program analysis," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 19, no. 11, 2009.

[11] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of IFIP'74*, Aug. 1974, pp. 471–475.

[12] Thomas M. Parks, *Bounded Scheduling of Process Networks*, Ph.D. thesis, Berkeley, Berkeley, CA, USA, 1995.

[13] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM New York, NY, USA, 1977, pp. 238–252.

[14] G. Stitt and F. Vahid, "New decompilation techniques for binary-level co-processor generation," in *IEEE/ACM International Conference on Computer-Aided Design, 2005. ICCAD-2005*, 2005, pp. 547–554.

[15] C.G. Nevill-Manning and I.H. Witten, "Identifying hierarchical structure in sequences: A linear-time algorithm," *Journal of Artificial Intelligence Research*, vol. 7, no. 1, pp. 67–82, 1997.

[16] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, 1991.

[17] Naoyuki Tamura, "Cream: Class Library for Constraint Programming in Java," .

[18] C. Zebelein, J. Falk, C. Haubelt, and J. Teich, "Classification of General Data Flow Actors into Known Models of Computation," *Proc. MEMOCODE, Anaheim, CA, USA*, pp. 119–128, 2008.