LUND UNIVERSITY

Static Analysis and Transformation of Dataflow Multimedia Applications

von Platen, Carl; Eker, Johan; Nilsson, Anders; Årzén, Karl-Erik

2012

Document Version:
Publisher's PDF, also known as Version of record

Link to publication

Citation for published version (APA):
von Platen, C., Eker, J., Nilsson, A., & Årzén, K.-E. (2012). *Static Analysis and Transformation of Dataflow Multimedia Applications*. (Technical Reports TFRT-7626). Department of Automatic Control, Lund Institute of Technology, Lund University.

Total number of authors:
4

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

ISSN 0280-5316
ISRN LUTFD2/TFRT--7626--SE

Static Analysis and Transformation of Dataflow Multimedia Applications

Carl von Platen
Johan Eker
Anders Nilsson
Karl-Erik Årzén

Lund University
Department of Automatic Control
November 2012

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
Internal report

Date of issue
November 2012

Document Number
ISRN LUTFD2/TFRT--7626--SE

Author(s)
Carl von Platen Ericsson Research, Sweden
Johan Eker, Ericsson Research, Sweden
Anders Nilsson, Dept. of Automatic Control, Lund University, Sweden
Karl-Erik Årzén, Dept. Of Automatic Control, Lund University, Sweden

Supervisor

Sponsoring organization

Title and subtitle
Static Analysis and Transformation of Dataflow Multimedia Applications

Abstract
An approach for merging statically schedulable subregions in dataflow models is presented. The approach combines abstract interpretation, loop analysis, and static scheduling of cyclo-static dataflow networks. The approach has been implemented in a Java-based tool that performs automatic classification of dataflow actors, generation of static schedules using constraint programming, and automatic merging of the finegrained actors in the subnetwork into a single, larger-grained actor. The approach is applied to an MPEG-4 SP video decoder implemented in the dataflow actors language CAL.

Keywords

Classification system and/ or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-13

Recipient's notes

Security classification

Static Analysis and Transformation of Dataflow Multimedia Applications

Carl von Platen, **Johan Eker**
Ericsson Research
Lund, Sweden
Anders Nilsson, **Karl-Erik Årzén**
Lund University
Lund, Sweden

***Abstract*—An approach for merging statically schedulable subregions in dataflow models is presented. The approach combines abstract interpretation, loop analysis, and static scheduling of cyclo-static dataflow networks. The approach has been implemented in a Java-based tool that performs automatic classification of dataflow actors, generation of static schedules using constraint programming, and automatic merging of the fine-grained actors in the subnetwork into a single, larger-grained actor. The approach is applied to an MPEG-4 SP video decoder implemented in the dataflow actors language CAL.**

I. Introduction

Programming for multicore systems is in many ways different from developing for unicore architectures. A particularly difficult problem is to make the performance scale with the number of cores. Silicon vendors have been extremely successful for decades in developing new generations of hardware that is both faster and backwards compatible. To achieve the same for multicore systems, applications need to be written such that they make use of an increasing number of cores: the full potential parallelism within an application must be exposed. The partitioning of tasks and functionality onto a specific set or cores must also not be a part of the implementation, but rather a step in the deployment phase. This calls for a programming model that is fine granular and allows the application to be partitioned in a number of different configurations depending on the target system at hand. We think that dataflow programming is a strong candidate for this and in this work we explore the possibilities to design scalable, yet efficient software for multicore hardware.

A dataflow model consists of components, called actors, which are connected by FIFO channels (see Fig 1). Each actor implements a particular computation, it operates on the data that it receives on its inputs and produces results that are transmitted to other actors via the FIFOs that are connected to its outputs. Theoretically this simple programming model has the power of expressing any kind of computation, but particularly applications in the signal processing domain are conveniently specified as dataflow models: media coding [1], image processing [2], embedded control [3], digital radio [4] and network processing [5] are examples.

Dataflow models expose parallelism naturally, since the actors interact only through the connecting FIFOs and may execute in parallel when sufficient input is provided. Both software and hardware can be synthesized from dataflow models [1]. Unlike an implementation in hardware, software implementation, which is the topic of this paper, is typically limited by the available parallelism of the target system. This means that parallel computations have to be serialized and, in the general case, scheduling decisions have to be taken dynamically, at run-time. On the one hand, we thus have a model of the program, which can be parallelized in a flexible, fine-granular manner. On the other hand, software implementations have to pay for this flexibility by run-time overheads.

The choice of granularity of a model is thus a matter of great concern. A fine granular model often promotes code reuse, since it may be composed of a number of standard library actors, and it also exposes parallelism to a higher degree, which makes it possible to target the code for hardware platform with a varying level of concurrency, ranging from ASIC/FPGA to unicore systems. Finally, it facilitates parallel development and testing. The latter can be an important factor in an industrial R&D teams. However, a more coarse grained model is likely to perform better due reduced communication and scheduling overhead. In this paper we show how to combine the best of both worlds by transforming a fine grained model into a more coarse grained one. The example models used in this paper for evaluation of the proposed methodology are taken from [1] and from the MPEG/ISO Reconfigurable Video Coding (RVC) standard [6], [7]. These models are fine granular and designed from a number of more generic actors with the ambition to support concurrent and pipelined implementations. For targets with low level of concurrence these models pose a challenge.

One solution to this problem is to impose specific restrictions on the behavior of the actors, so that the computations can be scheduled statically. It has been shown that highly optimized code can be synthesized from such models [8]. However, the restrictions that allow for static scheduling also reduce the expressiveness of the model; in particular it is generally not possible to express an actor that has varying rates of consumption and production, depending on the input it receives. Unfortunately, applications commonly have control-oriented parts, which introduce such input-dependent dynamism.

Rather than imposing restrictions on the model, the idea

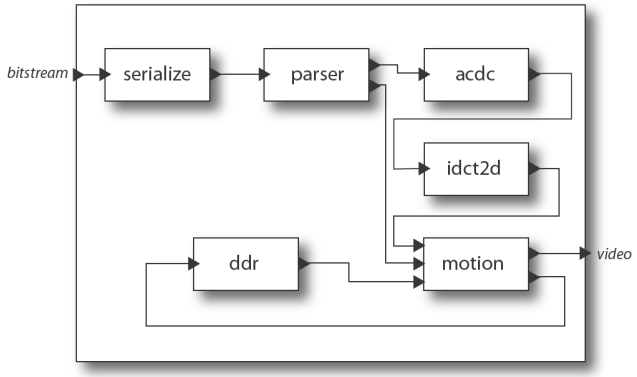

Fig. 1. A dataflow model consists of *actors* that are connected by *FIFOs*.

pursued in this paper is to use an expressive dataflow language in the specification of the model, to identify statically schedulable sub-networks using static analysis and to transform those sub-networks so that efficient code can be generated.

A tool with this purpose, the *ACTORS model compiler*, has been implemented. It consists of the following parts:

- actor classification,
- schedule generation, and
- actor merging,

together with a graphical user interface.

The *actor classifier* analyzes each actor of a dataflow model to decide if it can take part in a static schedule. If so, additional properties of the actor, which are needed to form a schedule, are computed. The *schedule generator* calculates a minimal repetitive sequential schedule of a given sub-network. In general, several schedules are admissible and a certain optimization criterion, such as the minimization of the required buffer space, is used to select one of the schedules. The schedule is realized by the *actor merger*, which combines the actors of the sub-network into a single, larger-grain actor, whose computations are serialized. In addition to eliminating dynamic scheduling decisions, the FIFOs that become internal to the merged actor are replaced by scalar variables or arrays.

One could say that this transformation trades flexibility for increased single-threaded performance. The merged actors can no longer be distributed over several processor cores, since the computations are serialized and the FIFO operations, which allow for inter-core communication, are removed. There is thus a trade-off between parallelization and reduction of run-time overheads.

A. Background

Dataflow programming has a long history beginning in the early 1970s with work of work Dennis [9] and Kahn [10] and has been the focus of much research since then, maybe most notably in relation to the Ptolemy project [11]. There exists a variety of dataflow execution models, which make different trade-offs between expressiveness and analyzability. Of particular interest are Kahn process networks [10], and synchronous dataflow networks (SDF) [12]. The latter is more constrained and allows for compile-time analysis for calculation of static schedules with bounded memory, leading to synthesized code that is particularly efficient. An interesting variant of SDF is cyclo static data flow (CSDF) [13]. An actor is SDF if the number of tokens consumed at each action firing from each input port and the number of tokens produced at each action firing to each output port are constant. Similarly, an actor is CSDF if the token consumption rate and production rate at each port follow a cyclic pattern. More general forms of dataflow programs are usually scheduled dynamically, which induces a run-time overhead.

The CAL language used in this work supports implementation of all types of actors, and has the nice property that it is possible at compile time to determine the category an actor belongs to, e.g. Kahn, SDF, CSDF, non-deterministic, etc., and also extract scheduling information.

B. Related Work

There is some ongoing parallel work in improving performance for CAL applications by statical scheduling of actors. In [14] the focus is on detecting SDF-like regions in CAL dataflow programs, by partitioning of actions using static analysis of the coupling between input ports, states variables, guards and output ports. This is then used for identifying sub-networks of actions that may be statically scheduled. This work was performed using an alternative CAL tool suite called ORCC [15], which lately has been extended with support for classification of actors. An approach presented in [16] to statically schedule actors is to use a parameterized variant of SDF to derive a quasi-static multiprocessor execution schedule. In [17] hierarchical SDF models are studied for the purpose of code generation. An SDF graph is encapsulated in an SDF actor, which may then be connected to other SDF actors to form a new SDF graph. Here the focus is on code generation for composite actors.

Code generation for multicore systems from dataflow graphs is the focus of [18], which also identifies the composition problems of SDF graphs, i.e. encapsulation of an SDF graph into an SDF actor may cause a deadlock unless the given environment is taken into account.

The use of constraint programming for scheduling and allocation of SDF graphs on multi-core platforms is presented in [19].

C. Outline

A short overview of CAL is given in Section II. The main parts of the model compiler and the theory behind it are presented in Section III. The results of applying the model compiler to a MPEG 4 Simple Profile decoder are presented in Section IV. The initial results show a speedup of around 18% for the approach.

II. The CAL Language

CAL [20] is a language a for writing dataflow actors. It has been used in a wide variety of applications and has been compiled to hardware and software implementations, and work on mixed HW/SW implementations is under way. CAL represents the basic components of a dataflow actor with firing

in a straightforward manner, providing structuring mechanisms that help the user to understand the functioning of an actor, and that aid tools to extract relevant information from an actor description at compile time. Recently a subset of CAL has been standardized by ISO/IEC MPEG (ISO/IEC IS 23001-4) and is used as the reference language for specification of MPEG video coding technology.

A simple example of a CAL actor is shown in the `Add` actor below, which has two input ports `p1` and `p2`, and one output port `p3`, all of type `int`. The actor contains one *action* (here tagged `a1`) that consumes one token on each input port, and produces one token on the output port. An actor may have one or several *actions*, which define when it is firable and the effect of such a firing. In this example, the port patterns imply that the actor is firable when there is at least one token on each input port.

```
actor Add () int p1, int p2 ⇒ int p3 :
  a1 : action [a], [b] ⇒ [a + b]  end
end
```

The actor above belongs to the SDF domain, since each time it fires it will consume one token at each of its input ports and generate one token at the output port. Detecting this automatically is trivial simply by inspection of the port patterns.

The slightly more advanced actor `AddSub` below has two actions, tagged `a1` and `a2`. The actor is fireable when there are at least two tokens on the input port `p1`. When it fires, it consumes two tokens and produces a single token on the output port `p2`. This actor is clearly also SDF, and again it is straightforward to detect so.

```
actor AddSub () int p1 ⇒ int p2 :

  a1 : action [a, b] ⇒ [a + b]
    guard a < b
  end

  a2 : action [a, b] ⇒ [a - b]
    guard a >= b
  end
end
```

The third example is more elaborate and shows a CSDF actor, which implements a simple algorithm that receives a package consisting of a header and a 64 token payload. The header defines how the payload shall be processed (either adding or subtracting all the elements). The actor has three state variable `counter`, `result`, and `type`. The execution order of the actions are governed by a finite state machine schedule `schedule fsm`, visualized in Figure 2. The actions either consume one token on the input port or produce one token on the output port. A closer look at signatures of the actions and the order in which they fire, determined by the guards and the fsm, shows that the this actor is indeed CSDF.

```
actor Receiver () int p1 ⇒ int p2 :
  int counter := 0;
  int result  := 0;
  int type    := 0;

  read_header  : action [t] ⇒
  do
    type := t;
  end

  read_payload1 : action  [t] ⇒
    guard counter < 64 and type = 1
  do
          add(t);
  end

  read_payload2 : action  [t] ⇒
    guard counter < 64 and type != 1
  do
          add(-t);
  end

  done : action ⇒ [result]
    guard counter = 64
  do
    counter := 0;
    result  := 0;
  end

  procedure add(int i)
  begin
    result := result + i;
    counter := counter + 1;
  end

  schedule fsm s0 :
    s0 (read_header)   --> s1;
    s1 (read_payload1) --> s2;
    s2 (read_payload1) --> s2;
    s2 (done) --> s0;
    s1 (read_payload2) --> s3;
    s3 (read_payload2) --> s3;
    s3 (done) --> s0;
   end

end
```

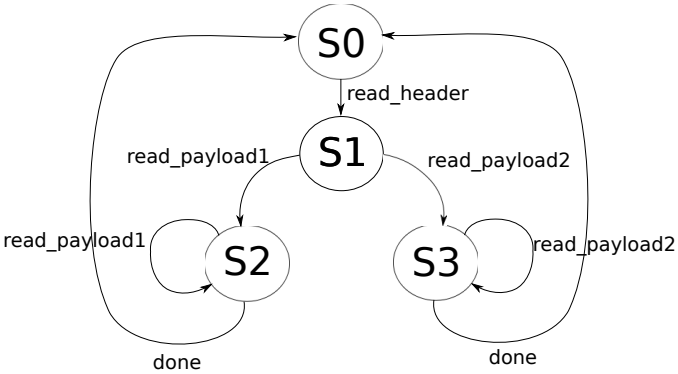

Fig. 2. The finite state machine from the actor `Receiver`

A third option to control the order in which the actions are fired is by priorities. In the `BiasedMerge` actor a priority order is established between the two actions labeled `A` and `B`. It ensures that in case both actions can fire, the one labeled `A` will be given preference over the one labeled `B`.

```
actor BiasedMerge () int p1, int p2 ⇒ int p3:
  A: action p1:[x] ⇒ p3:[x] end
  B: action p2:[x] ⇒ p3:[x] end
```

```
  priority A > B; end
end
```

For an in-depth description of the language, the reader is referred to the language report [20]. A large selection of example actors is available at the OpenDF repository,[1] among them the MPEG-4 decoder discussed below.

A. The NL Network Language

CAL actors are connected and instantiated using the Network Language, NL, which is a language for expressing algorithms that compute directed graphs among entities with ports. A small example is shown in the network `N` below, where the CAL actors described above are connected.

The NL supports hierarchical structures and a network can in itself have ports. It is also generative in the sense that entities may be instantiated and connected using comprehensions, loops, etc.

```
network N () In1, In2 ⇒ Out:

entities
  r  = Receiver();
  bm = BiasedMerge();
  as = AddSub();

structure
  In1 --> r.p1;
  In2 --> as.p1;
  r.p2  --> bm.p1;
  as.p2 --> bm.p2
  bm.p3 --> Out;
end
```

B. OpenDF Tool Chain

The OpenDF tool chain [21] used in this work consists of a simulator and compilers for hardware (FPGA) and software (ANSI C code). The CAL compiler frontend generates code on an intermediate format called XLIM [22], which is compiled into either VHDL or C code. XLIM is an XML based format that represents a program in static single assignment (SSA) form. The bodies of the actions are represented as a set of procedures and the guards, priorities, finite state machine, together with tests for token availability, are represented in a special action selection procedure called the *action scheduler*.

As part of the tool chain there is also an elaborator that compiles and flattens NL networks into the XDF, which is an XML based format for representing dataflow graphs.

The OpenDF tool chain is open source and released under the BSD license.

III. Model Compilation

The model compiler consists of three main parts: actor classification, schedule generation, and actor merging, each of which is explained in more detail in separate sections below. The merging is done after the actor network has been partitioned, and it is only static actors within the same partition that may be merged. The partitioning is performed using design space exploration techniques using objectives that, e.g., balance the computational load or minimize the amount of communication between partitions. The details of this are, however, outside the scope of this paper.

[1] http://www.opendf.net

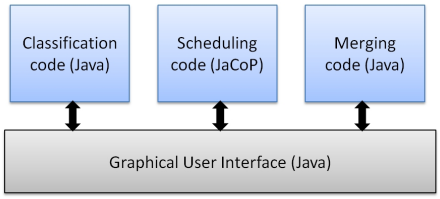

Fig. 3. The model compiler structure.

At several points in the merging process user intervention is required. For example, the user needs to select which actors to merge, which optimization criterion to use, and which of the generated schedules to use. Since dataflow networks are graphical in nature a graphical user interface is used to connect the three main parts together. The user interface is implemented in Java using Swing and the JGo graphics library [23]. The user interface takes a flat XDF file as input and displays it graphically using automatic layout. Interacting with the network using the mouse the user selects actors and invokes the classification, scheduling, and merging. The structure of the model compiler is shown in Fig. 3.

A. Actor Classification

Actor classification is the analysis in the Model Compiler, which determines the opportunities for static scheduling. Classification is closely related to the concept of (dataflow) computation models. Actor classification has the purpose of identifying actors that adhere to particular restrictions (such as those of SDF and CSDF), which allows the actors to be scheduled statically. Additional properties, particularly the rates at which an actor consumes and produces tokens, are computed as a side-effect.

1) Properties determined by actor classification: The actor classifier works by analyzing the internal behavior of each actor in isolation; the following properties are determined:

- Classification of the actor: “static” or “dynamic”,
- whether the actor executes indefinitely or if it may terminate, and
- a specification of the static firing sequence, provided that such a sequence was identified.

The classification is based on the sequence of actions, which an actor might fire. An actor is classified as “static” if the token rates can be determined beforehand and “dynamic” otherwise. In particular, an actor whose token rates depend on the inputs it receives falls into the “dynamic” class. Classification is conservative in the sense that unless a static firing sequence can be found, the actor is assumed to be “dynamic”. Any misclassification thus attributes the actor to a more general class than a perfect classifier would.

Actor classification also determines whether an actor is guaranteed to execute indefinitely or if it may enter a state,

from which no further firings are possible (termination). Again, the results are conservative: possible termination is assumed unless it can be ruled out.

In the case of "static" actors, results also include the specification of the firing sequence with the token production and consumption rates of each step of the sequence. In general, the produced static firing sequence consists of an initial sequence, which is executed once, and/or a periodic sequence that is repeated indefinitely.

2) Action selection: The execution of an actor is performed in steps, known as *firings*, in which one of the actor's actions is executed. The action selection is based on the internal state of the actor, the availability of inputs and, possibly, the value of inputs.

Action selection can be represented by a decision diagram (see Fig 4a), in which the interior nodes correspond to tests and the leaves correspond to action firings. There may also be a leaf that corresponds to termination. The leaves that correspond to action firings have side-effects: mutation of the internal state and the consumption/production of inputs/outputs, all other nodes are side-effect free.

The decision diagram can be viewed as a control-flow graph, which is repeated indefinitely (or rather: until the terminal leaf is reached). Repetition is indicated by the dashed arrows in Fig 4a.

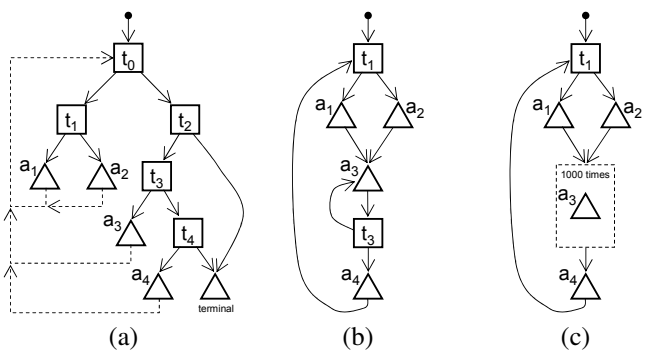

Fig. 4. Action scheduler represented as (a) decision diagram with added back-edges (dashed), (b) a control-flow graph with (some) infeasible paths removed and (c) a control-flow graph, in which a loop has been summarized into a single node.

3) Removal of infeasible paths: The initial incarnation of the control-flow graph generally contains infeasible paths. Not to get overly pessimistic results, a more precise graph is needed.

Two techniques are used to remove (some of) the infeasible paths: partial evaluation of the tests and loop analysis. The search for a static firing sequence is made in the resulting control-flow graph.

First, the state space of the actor is enumerated using abstract interpretation. As a result, we get an abstraction of the actor's internal state that may reach each leaf (each action). Using this information we can reduce the set of potential successors and get a more precise control-flow graph, like the one shown in Fig 4b. If the terminal leaf was reached during the enumeration, we must assume that the actor may terminate. Otherwise, we know that the actor will execute indefinitely.

The implementation of state-space enumeration does not depend on the choice of abstract domain, but we presently use only *integer intervals* $[a, b]$. This is a good abstraction for "normal" arithmetic operations (+,-,*,/ etc.) and relational operators, and we note that constant values are propagated as a special case (intervals $[c, c]$). However, considerable loss of precision is caused by other operations (such as bitwise and, or, xor).

State-space enumeration is performed in repeated passes over the decision diagram. The number of required iterations depends on properties of the actor and the abstract domain, in which the state is represented. So called *widening* [24] of the abstract state is required to make very "tall" domains, such as integer intervals, practical. Otherwise, an essentially unbounded number of iterations might be needed to reach a fixed point.

In a second step, precision is further improved by detecting loops with constant trip-counts. The present implementation is based on detection of natural loops and induction variables, which are standard techniques (e.g. see [25]). Fig 4c illustrates the case of an inner loop, whose trip-count could be determined statically, and an infinite outer loop.

4) Finding static firing sequences: We say that an actor has a static firing sequence if the actions, which might fire in any given step of the sequence, consumes and produces the same number of tokens. Thus, one way of classifying the actor would be to simply enumerate the sets of admissible actions for each step of the firing sequence and check the token rates. Although this is essentially the approach taken, there are two complications: the firing sequences are generally infinite and it might not be possible to decide the exact set of admissible actions.

One way of estimating the sets of admissible actions is to consider reachability in the control-flow graph. For instance, in the graph of Fig 4b, the following actions are reachable in the first five steps:
$A_1 = \{a_1, a_2\}$,
$A_2 = \{a_3\}$,
$A_3 = \{a_3, a_4\}$,
$A_4 = \{a_1, a_2, a_3, a_4\}$,
$A_5 = \{a_1, a_2, a_3, a_4\}$,
...

Each new set is formed by the successors (see Fig 5) of the actions in the set of the preceding step in the sequence. This means that even an infinite firing sequence leads to a finite enumeration of sets; a recurring set (such as $A_4 = A_5$ in the above example) will eventually be found. Such a recurrence marks a subsequence that is repeated indefinitely.

It remains to check the token rates. Since A_4 contains all four actions, we find the firing sequence to be "static" only in the case of an SDF actor (whose actions have the same token rate). Any difference in token rates would render the actor a "dynamic" classification.

In contrast, if we start out from the control-flow graph of Fig 4c, in which the inner loop is represented as a single node, it would be sufficient that actions a_1 and a_2 had the same rates, since we get the following, more precise, estimation of the admissible actions:

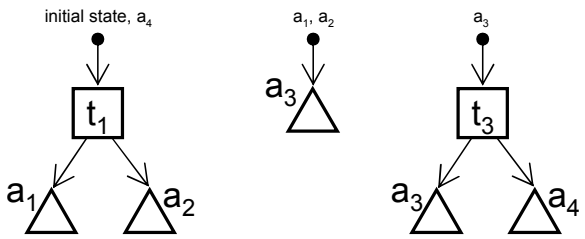

Fig. 5. Successors: the sets of actions that may immediately follow the initial node and each of the actions (given the control-flow graph of Fig 4b)

$A_1 = \{a_1, a_2\}$,
$A_2 = \{a_3\}$,
...
$A_{1001} = \{a_3\}$,
$A_{1002} = \{a_4\}$,
$A_{1003} = \{a_1, a_2\}$,
...

Explicit enumeration of this long sequence is avoided by analyzing loop nests bottom-up. In this particular case, the static firing sequence of the inner loop (1000 times action a_3) is determined first, later to be combined with the static firing sequence of the outer loop; thus arriving at a static firing sequence with a period of 1002. The resulting firing sequence is represented compactly as a *looped schedule* [26], in which repetitions within the sequence are factored out.

B. Schedule Generation

The result of the classification is displayed graphically to the user by using different colors for the actors. Green actors have been classified as statiic, whereas yellow and red indicate different classes of dynamic actors. All sub-networks consisting only of static actors that all belong to the same partition are candidates for merging. It is the task of the user to select which sub-network to merge.

The objective of the schedule generation is to generate a *periodic admissible sequential schedule (PASS)* for the connected sub-network, henceforth referred to simply as the network, in which all actors are SDF or CSDF. The theory behind this was originally derived in [12] for the case of SDF networks and then generalized to CSDF networks in [27]. Here a brief overview of the CDSF case is given.

1) Scheduling of CSDF networks: The starting point for the scheduling is the topology, or incidence, matrix Γ of the network. The topology matrix is a MxN-matrix where M is the number of connections (edges) between the nodes (actors) in the network and N is the number of nodes. Two nodes can be connected by multiple edges corresponding to connections between different ports. We define σ_{ij} as the sum of the rates in the token pattern associated with the connection between node j and connection i, and p_{ij} as the length of this token pattern. A token pattern is allowed to contain zero elements, e.g., 1,0,3. Here this means that when the node is executed for the second time no tokens are produced (or consumed) at this connection. In a CSDF network the ith actor will fire in a cycle with a period P_i equal to the least common multiple (lcm) of the lengths of all the token patterns of the node. With these definitions the (i, j)th entry in the topology matrix is given by $P_j \sigma_{ij}/p_{ij}$. If node j consumes tokens from arc i, the value is negative.

The first step in finding the schedule is to calculate the *repetition vector* r for the network, which is given by the smallest integer solution to the equation

$$\Gamma r = \mathcal{O}, \tag{1}$$

where $\mathcal{O}$ is a column vector of length N of zeros. The elements in r are the number of cycles that each node should fire. The corresponding actor repetition vector q is obtained by multiplying each element in r with the corresponding P_i value.

Given the repetition vector finding a schedule consists of finding an admissible sequence of node firings, i.e., such that if the nodes are fired according to the sequence the amount of tokens in the buffers will remain nonnegative and bounded, that meets the constraints imposed by the repetition vector, and which minimizes some desired objective. This can be stated as a search problem.

2) Constraint Programming Search: The calculation of the repetition vector and the schedule generation are formulated as search problems using constraint programming with JaCoP. JaCoP (Java Constraint Programming) is a constraint solver engine developed for constrained finite domain variable problems, e.g., integer and boolean problems, [28]. The search problems are modeled as a set of constraints over integer variables. The constraints are given as arithmetic expressions, equalities, inequalities, etc. Depth-first branch-and-bound search techniques are then used together with constraint consistency techniques to find solutions which satisfy the constraints and optimize a given cost function.

Constraint Formulation for Repetition Vector
The search for the repetition vector is formulated using JaCoP as follows. The r vector is defined as a vector of length N of integers in the range $[1, MAX_int]$. The upper and lower bounds act as implicit constraints on the values of r. The balance equation $\Gamma r = \mathcal{O}$ is represented by M weighted sum constraints, each constraint expressing that the sum of the elements in the corresponding row i of Γ weighted by the elements in r should be equal to zero, i.e. $r_1\Gamma_{i1} + r_2\Gamma_{i2} + \cdots + r_n\Gamma_{in} = 0$.

In order to find the smallest solution a new integer variable, $Rsum$, is defined together with a sum constraint stating that $Rsum = \sum_i r_i$. The search is defined as a depth-first branch-and-bound search with $Rsum$ as the cost function that should be minimized. Each time a solution with $Rsum = value$ is found a new constraint $Rsum < value$ is imposed, causing the search to find solutions with lower cost until no more solutions can be found, proving that the last found solution is optimal. The variables searched for are the elements of r and during the search the values are assigned starting at the smallest value in the domain.

Constraint Formulation for Scheduling
The inputs to the schedule generation consist of the actor repetition vector q and a MxN-matrix G where the elements are vectors containing the corresponding port token patterns. The length of the schedule, i.e., the sum of the elements in q is denoted S. The main search variables are contained in

the matrices x and b. The x matrix indicates which actor that is fired at each step in the schedule and has dimension SxN. The value of $x[s][n]$ is one if actor n is fired at step s and zero otherwise. The b matrix contains the size of all the buffers for each step in the schedule and has dimension SxM. In addition to this the matrix $xCumul[S][N]$ is used for storing the cumulative sum of the elements in x and the matrix $maxBufs[M]$ contains the maximum size needed for buffer M during the schedule.

The main constraints are the following.

- Constraints specifying that only one actor may fire at each step, i.e., that the sum of the values in each row of x is one.
- Constraints that update the elements in $xCumul$ based on values in x.
- Constraints that ensure that the values in the last row of $xCumul$ equal the values in the repetition vector q.
- Constraints that specify that the initial value of each buffer is 0.
- A set of constraints that together ensure that each element in the buffer matrix b is updated according to the topology matrix G. These constraints use $xCumul$ to decide the actual number of tokens produced and consumed at each step, based on the port token patterns in G.
- Constraints for updating $maxBufs$.

The optimization currently supports three optimization objectives.

- It is possible to search for all admissible schedules.
- It is possible to search for the schedules which minimize the size of the largest internal buffer.
- It is possible to search for the schedules which minimize the sum of the maximum size of all the internal buffers, i.e. the total buffer space.

When searching for all admissible schedules the problem is stated as a depth-first search problem. The variables searched for are the elements of x and $maxBufs$. During the search values are assigned starting at the smallest value in the domain. The search is setup to search for all the solutions that fulfill the constraints.

When searching for schedules that minimize the buffer size first a new search variable is defined corresponding to the entity that should be minimized, together with the new constraints required to specify this. Then a search is performed that finds a solution that minimizes the cost. However, in most cases there are several schedules that have the same minimum buffer requirements. In order to find all of these a new constraint is added that specifies that the value of the cost function should equal the optimal value found and a new search is performed for all the solutions fulfilling the constraints.

Also for modest size problems the number of schedules can be very large. Also, the problem of finding a minimal buffering for a live execution has been shown to be NP-complete, [29]. Therefore the search is always performed with a user-defined timeout with a default value of 10 sec. When the timeout expires all solutions found up to that point are returned. The user can then decide to either select one of the schedules found, perform a new search with a longer timeout, or to change the actor sub-network for which the scheduling is performed.

The output of the scheduling is presented to the user as a list of schedules and the associated maximum size of each internal buffer, in a pop-up window. The user then selects one of these schedules in order to perform the actual merging.

C. Actor Merging

Actor merging is performed on the intermediate XDF/XLIM level, where all actors in a network have been parametrized and instantiated. It is then safe to perform transformations on the actors forming the network with no risk for unwanted side effects due to certain actors being instantiated more than once at several locations in the network.

The merging conceptually consists of creating a new XLIM file corresponding to the merged actor and updating the network desctription file (XDF). The XLIM file contains the actions of all the actors that are merged together, and a new action scheduler that corresponds to the static schedule.

The input to the actor merging comes both from the schedule generator and from the classifier. From the schedule generator comes the sub-network to be merged, the schedule according to which it should be merged, the maximum size of all internal buffers in the merged actor, the number of tokens required at each input port of the merged actor in order for it to be firable, and the number of empty slots needed in the FIFOs attached to each output port in order to fire the actor. From the classifier the actor merger ideally receives information about which action that should be fired at each phase for all the actors involved in the schedule. If that cannot be decided uniquely off-line the classifer outputs the corresponding part of of the action scheduler decision tree that, when evaluated at runtime, decides which action to be executed. This decision logic is then inserted at the appropriate place in the generated action scheduler.

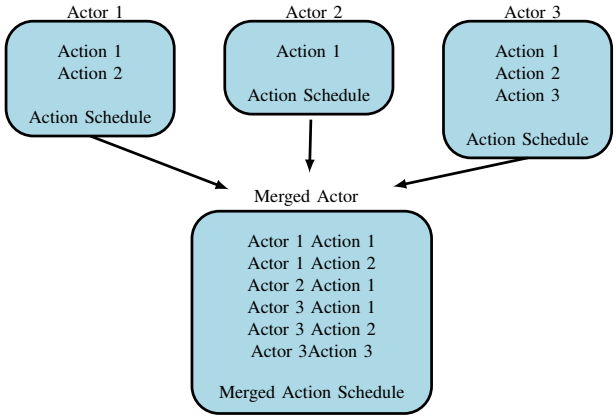

Fig. 6. Merging three actors into one.

The procedure for merging actors is as follows. First collect all actions from the actors to be merged and put them as actions in a new actor, see Fig. 6. Then search all connections in

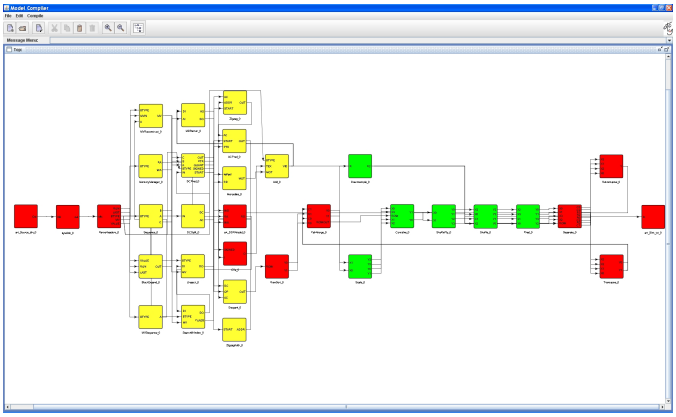

Fig. 7. Classified MPEG 4 SP decoder: the green actors are "static", yellow and red indicate different classes of "dynamic" actors.

the network for connections between the actors to be merged. If a connection between two actors is found, a corresponding circular buffer is created in the new actor and writes, reads, and peeks on the FIFO queue are replaced with accesses to this buffer. Then the connection is removed from the network description. If the maximum size of an internal buffer is 1 then a simple scalar state variable is used instead of a circular buffer. This speeds up the execution considerably.

The actor merging is implemented using the state-of-art compiler construction tool, JastAdd [30]. Compilers for both the XDF and XLIM intermediate formats have been developed. Using the aspect-oriented features of JastAdd new functionality can then be added to the compilers in a modular fashion.

IV. Results

In order to evaluate the model compiler a CAL model of an MPEG-4 Simple Profile video decoder was used. The model was developed at Xilinx to demonstrate FPGA code synthesis. It is published along with the OpenDF tools on sourceforge [31]. Since merging can only be applied to actors within the same partition a single partition is used for the entire decoder in the evaluation.

The runtime system is non-preemptive and statically partitioned. Each core is assigned a ordered list of actors that are executed in a round robin fashion with the goal to minimize scheduling overhead and maximize throughput. When an actor is fired it will continue to repeatedly fire as long as there are input tokens available and space left on the output FIFO:s. The runtime systems is described in detail in [**?**] and the source code for the runtime system is freely available at [31].

The output from the classification as seen through the graphical interface is shown in Figure 7. The green actors have been automatically classified as static and are hence candidates for merging. The yellow and red actors have been classified as having different variants of non-static behavior. Hence, for the actor merging the key is to find sub-networks composed only of green actors. Five out of the six static actors form a static sub-network that implements a 1-dimensional inverse discrete cosine transform (idct1d). The actors involved here are: *Scale, Combine, Shufflefly, Shuffle*, and *Final*. The schedule length for idct1d is 17 and there are 4264 admissible static schedules. However, of these only 44 minimize the internal buffer space required. For these, all the 14 internal buffers have a maximum size of 1 and can, hence, be represented as scalar variables in the merged actor.

The speedup when decoding a video sequence without and with merging the idct1d actors into a single actor is 18%, measured in terms of increased frame rate. However, the idct1d only constitues one small part of the complete decoder. Isolating the idct1d part as a separate test case reveals that the speedup gained from merging these five actors is 300%.

V. Conclusions

The work presented in this paper demonstrates that it is not only possible, but also practical, to identify and specialize sub-networks with static behavior within a dynamic dataflow model. Well-known techniques are combined in the analysis of a model: abstract interpretation [24], traditional loop analysis [25] and static scheduling of CSDF networks [12], [32]. A static schedule is realized by merging a sub-network of fine-grained actors into a single, larger-grain actor. The computations of the sub-network are serialized, by which dynamic scheduling decisions are eliminated. Further, communication within the sub-network is specialized by substituting access to variables for more costly FIFO operations.

This means that the model can be specified in a fine-granular fashion, using the full expressive power of a dataflow language like CAL while generating specialized code for each sub-network with static behavior that was mapped to a single processor core. It is thus possible to specify a model that is portable over a wide range of target architectures and specialize it given a particular target.

The effectiveness of the model compiler was assessed using a model of an MPEG-4 SP video decoder. By specializing a statically schedulable sub-network of the model, a unicore implementation was speed-up by 18%.

Actor classification can also provide feedback to the programmer. Given that the performance of an implementation can be improved significantly, there is a strong incitement to use actors with static behavior when possible. We found several cases of unnecessary input-dependence in the MPEG-4 SP decoder. The model could be refactored so that larger sub-networks of statically schedulable actors would result.

A. Future work

The current implementation of schedule generation does not consider the structure of the surrounding network when selecting a static schedule of a sub-network. When surrounded by a cyclic dataflow path, a statically scheduled sub-network might cause deadlock. The present, unsatisfactory, solution is to allow the programmer to control the selection of schedule or avoid merging potentially problematic sub-networks altogether. In an evolved implementation, we intend to use additional constraints [18], [17] to guide schedule selection so that deadlock is avoided.

Further, we intend to extend the scope of the model compiler. The current approach is based on analysis of

each actor in isolation and model transformation is limited to sub-networks of statically schedulable actors. Additional specialization is possible by considering also the network structure, such as the statically schedulable regions of [14]. Other techniques address the scheduling overhead of dynamic dataflow, such as quasi-static scheduling (e.g. see [16]). Such generalization fits within the framework of the model compiler and presents a promising direction for future work.

B. Acknowledgements

The work has been done with partial support from the EC FP7 project ACTORS (Contract IST-216586) and from the VINNOVA/Ericsson project "Feedback Based Resource Management and Code Generation for Real-Time Systems".

REFERENCES

[1] J. Janneck, I. Miller, D. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing hardware from dataflow programs: An MPEG-4 simple profile decoder case study," in *Proceedings of IEEE Workshop on Signal Processing Systems*, 2008.

[2] 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.

[3] S. S. Bhattacharyya and W. S. Levine, "Optimization of signal processing software for control system implementation," in *Proceedings of the IEEE Symposium on Computer-Aided Control Systems Design*, Munich, Germany, October 2006, pp. 1562–1567, invited paper.

[4] T. Olsson, A. Carlsson, L. Wilhelmsson, J. Eker, C. von Platen, and I. Diaz, "A reconfigurable OFDM inner receiver implemented in the CAL dataflow language," in *ISCAS*, 2010, pp. 2904 – 2907.

[5] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "The click modular router," *SIGOPS Oper. Syst. Rev.*, vol. 34, no. 2, pp. 24–25, 2000.

[6] "Information technology – mpeg video technologies – part 4: Video tool library," Moving Pictures Expers Group (MPEG), Tech. Rep. ISO/IEC 23002-4:2010, 2009. [Online]. Available: http://www.iso.org

[7] "Information technology – mpeg systems technologies – part 4: Codec configuration representation," Moving Pictures Expers Group (MPEG), Tech. Rep. ISO/IEC 23001-4:2009, 2009. [Online]. Available: http://www.iso.org

[8] S. S. Battacharyya, E. A. Lee, and P. K. Murthy, *Software Synthesis from Dataflow Graphs*. Norwell, MA, USA: Kluwer Academic Publishers, 1996.

[9] J. B. Dennis, "First version of a data flow procedure language," in *Symposium on Programming*, ser. Lecture Notes in Computer Science, B. Robinet, Ed., vol. 19. Springer, 1974, pp. 362–376.

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

[11] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity—the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 2, 2003.

[12] E.A.Lee and D.G.Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, Jan 1987.

[13] G. Bilsen, R. L. M. Engels, and J. Peperstraete, "Cyclo-static data flow," *IEEE Int. Conf. ASSP*, pp. 3255–3258, May 1995.

[14] R. Gu, J. W. Janneck, M. Raulet, and S. S. Bhattacharyya, "Exploiting statically schedulable regions in dataflow programs," *Journal of Signal Processing Systems*, 2010.

[15] "Open RVC-CAL Compiler." http://sourceforge.net/projects/orcc/.

[16] J. Boutellier, C. Lucarz, S. Lafond, V. Gomez, and M. Mattavelli, "Quasi-Static scheduling of CAL actor networks for reconfigurable video coding," *Journal of Signal Processing Systems*, 2009. [Online]. Available: http://dx.doi.org/10.1007/s11265-009-0389-5

[17] S. Tripakis, D. Bui, M. Geilen, B. Rodiers, and E. A. Lee, "Compositionality in synchronous data flow: Modular code generation from hierarchical sdf graphs," http://www-verimag.imag.fr/ tripakis/publis.html, 2010.

[18] J. Pino, S. Bhattacharyya, and E. Lee, "A Hierarchical Multiprocessor Scheduling Framework for Synchronous Dataflow Graphs," EECS Department, University of California, Berkeley, Tech. Rep. Technical Report UCB/ERL M95/36, 1995.

[19] A. Bonfietti, L. Benini, M. Lombardi, and M. Milano, "An efficient and complete approach for throughput-maximal SDF allocation and scheduling on multi-core platforms," in *Proceedings of DATE 2010*, March 2010.

[20] J. Eker and J. Janneck, "CAL Language Report," University of California at Berkeley, Tech. Rep. ERL Technical Memo UCB/ERL M03/48, Dec. 2003.

[21] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet, "OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems," *SIGARCH Comput. Archit. News*, vol. 36, no. 5, pp. 29–35, 2008.

[22] J. Janneck, "Xlim - an xml language-independent model," http://opendf.svn.sourceforge.net/viewvc/opendf/doc/Xlim/XLIM.pdf, 2007.

[23] JGo, "http://www.nwoods.com," URL, 2010.

[24] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. New York, NY, USA: ACM, 1977, pp. 238–252.

[25] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, principles, techniques, and tools*. Addison Wesley, 1985.

[26] S. Bhattacharyya and E. A. Lee, "Looped schedules for dataflow descriptions of multirate DSP algorithms," EECS Department, University of California, Berkeley, Tech. Rep. UCB/ERL M93/37, 1993. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1993/2346.html

[27] T. M. Parks, J. L. Pino, and E. A. Lee, "A comparison of synchronous and cyclo-static dataflow," in *Proceedings of the Asilomar Conference of Signals, Systems and Computers*, October 1995.

[28] K. Kuchcinski, "Constraints-driven scheduling and resource assignment," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 8, no. 3, pp. 355–383, 2003.

[29] P. Murthy, "Scheduling techniques for synchronous and multidimensional synchronous dataflow," Ph.D. dissertation, University of California at Berkeley, USA, Tech. Rep., 1996.

[30] T. Ekman and G. Hedin, "The JastAdd System - modular extensible compiler construction," *Science of Computer Programming*, vol. 69, pp. 14–26, October 2007.

[31] "Open DataFlow Sourceforge Project." http://opendf.sourceforge.net/.

[32] A. Blomdell, K.-E. Årzén, and C. Chen Xu, "OpenDF extensions," Tech. Rep.

[33] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static dataflow," *IEEE Trans. Signal Processing*, vol. 44, no. 2, pp. 397–408, 1996.