SCHEDULING OF DYNAMIC DATAFLOW PROGRAMS BASED ON STATE SPACE ANALYSIS

Johan Ersfolk[1,3]*, Ghislain Roquier*[2]*, Johan Lilius*[1]*, Marco Mattavelli*[2]

[1]Åbo Akademi University, Finland
[2]École Polytechnique Fédérale de Lausanne, Switzerland
[3]Turku Centre for Computer Science, Finland

ABSTRACT

Compile-time scheduling of dynamic dataflow programs is still an open problem. This paper presents how scheduling of dynamic portions of asynchronous dataflow networks described using CAL language can be determined before execution by the analysis of the state space of network partitions. Experiments show that the number of run-time operations employed by dynamic schedulers is largely reduced when the schedules extracted by the state analysis are employed.

Index Terms— Dataflow programming, model checking, MPEG-4 decoder

1. INTRODUCTION

Quasi-static scheduling of network partitions has been proposed for the scheduling of dynamic dataflow programs [1, 2]. In dynamic dataflow programs, scheduling decisions are determined at run-time which introduce a significant overhead. Quasi-static scheduling intends to make most of the scheduling decisions at compile-time by determining most of the schedule statically while leaving only the necessary scheduling decision for run-time. The main benefit is that the run-time scheduling overheads can be reduced whereas the programming languages used maintain a high level of expressiveness to naturally and efficiently specify dynamic behaviors of algorithms.

Dynamic behavior appears in most applications. In a video decoder, for instance, the operations to be performed depend on the type of data that flows through the program. The behavior of an actor in a dataflow network may depend on its current and previous input values as well as its internal state, making scheduling a run-time problem. However, the idea pursued here is that when a network partition including several actors is analyzed as a whole, most of the checks required to execute a single actor becomes redundant. This is what is usually done by analyzing token rates and applying it to static dataflow network sections, however, the formal description of an actor network behavior provided by CAL [3] language lets us go beyond token rate analysis and enables us to explore the whole state space of the network partition.

This paper presents how such analysis can yield schedules that well represents the execution of dynamic portions of dataflow networks specified using CAL. The scheduling approach uses a model checker to analyze the state space of a CAL sub-network, the CAL sub-network is converted to an equivalent Promela program and analyzed using the SPIN [4] model checker, identifying deterministic schedules that link recurring network execution states. Therefore, the only dynamic operations that the scheduler needs to execute at runtime are the guard evaluations between states linked by the obtained deterministic schedules.

In this paper we describe a method that extract information from a CAL program so that a model checker can generate an appropriate model that can be use to find deterministic schedules.

2. BACKGROUND AND RELATED WORK

A CAL program consist of a set of actors exchanging data tokens by unidirectional order preserving channels virtually of infinite size (i.e. FIFOs). The actors execute the program by firing eligible actions. An action is eligible if: 1. tokens are available 2. its guard expression (including any state predicate) evaluates to true, and 3. it is enabled by the action scheduler 4. it has a higher priority if more actions are enabled by the action scheduler in that state. The action scheduler operator, if used, express, in the form of finite state machine (FSM) transitions, the actor behavior in terms of action eligibility. An action is only fired if the current state has a transition corresponding to that action. Each action may consume and/or produce tokens from one or more input or output port connected to the FIFO channels (an action may also have no input or output). A subset of CAL [3], named RVC-CAL, has been standardized by ISO/IEC MPEG [5, 6]. In MPEG a video decoder is described by a network configuration of RVC-CAL actors instantiated from a standardized actor library.

Other types of approaches trying to schedule CAL networks more efficiently exist. In [7] CAL actors are classified to determine the Model of Computation (MoC) an actor belongs to. By using abstract interpretation and static analysis

of the CAL program, it is possible to identify if the actor can be classified to belong to synchronous dataflow (SDF), cyclo-static dataflow or parameterized dataflow, which can be scheduled more efficiently. Another interesting approach which is presented in [2], automatically detects statically schedulable regions within CAL programs. This approach finds SDF like regions that can be statically scheduled while leaving dynamic parts for the run-time scheduler. In [8] and [9], CAL networks are scheduled for given groups of input values associated with a control token. While [8] uses a model checker to extract schedules, [9] uses dynamic code analysis which identifies static schedules for the different values of the control tokens passing through the network. In this paper we improve the approach presented in [8] by defining which information is relevant for scheduling and defines the state of the dataflow network. Whereas in [8] we only could assure that a check on the value of an actors input port could be omitted in a number of special cases, we can now omit any such check, however, for some cases this might result in a very large number of schedules.

3. STATE SPACE ANALYSIS

The scheduling approach described here uses a model checker to analyze the state space of a CAL sub-network. The CAL sub-network is converted to an equivalent Promela program and analyzed using the SPIN model checker identifying deterministic schedules. A more detailed description of how the Promela program is constructed is available in [8], we will here only concentrate on how the appropriate information is collected and used to generate schedules.

The deterministic schedules to be generated can be described as lists of action firings where the appropriate list is chosen depending on a condition that must be checked at run-time. The information that must be used to yield schedules that behave correctly at run-time includes the current state of the dataflow network as well as conditions regarding the inputs to the network. As the state space of a typical CAL program, such as a video decoder, is very large, only the information relevant to the scheduling should be considered in the model, whereas other data should only be seen as valueless tokens. To achieve this objective, the program representation must be analyzed to extract only the relevant information that must be included in the derived Promela program.

3.1. Extracting Scheduling Related Information

For the scheduling of a single actor, the information needed is the state of the actor's FSM and the values of any variable or input port used in a guard. When we schedule a network of actors, the goal is to avoid unnecessary checks of input values to individual actors. Instead we analyze how control tokens propagate through the network and identify every variable related to generating such values. These variables will be part of the network state used in the model checker as they clearly affect the scheduling.

The first step in the analysis is to describe the dependencies between variables in the program. This can be described as a binary relation, $R \subseteq V \times V$, which is a subset of all the possible pairs of variables in the actor such that $(x, y) \in R$ indicates that variable x depends on variable y. Building such a relation at compile-time is simple as we need to read each instruction only once. Furthermore, variables are not considered to depend on the guard expression of the action in which the variable is assigned, the reason for this is that the guard is part of the scheduling of the actor and not of the data flow through the actor. If-statements and loops, on the other hand, are considered to be part of the variable dependencies as these are not visible on the scheduling of an actor, but only affect the values written to variables and ports.

This analysis has been implemented in the Orcc compiler [10], which means that we work with the intermediate representation (IR) of this compiler. What is relevant for this discussion is the that actions consist of assignments, loops and if-statements. Actions have local variables which can read or write global variables using load and store instructions; only the global variables keep their value between action firings. The local variables are not important for the analysis, but are only part of the chain of variables that link the global variables together. Building R from such instructions is trivial. Load and store instructions and assignments have a left hand side consisting of a variable and a right hand side which is an expression. The variable on the left hand side depends on each variable on the right hand side; for each variable on the right hand side we therefore add one variable relation. If an instruction resides in a block belonging to a if or while statement, it will also depend on each variable used in the condition.

The compiler is used to produce the following information regarding variables and ports. We have both global and local variables, $V \equiv V_g \cup V_l$, and input and output ports, $P \equiv P_i \cup P_o$. Furthermore, as some global variables correspond to ports we have $P \subseteq V_g$. By looking at the relation R which describes how variables depend on each other we can find some basic properties of the variables.

An output port $p \in P_o$ depends on an input port $q \in P_i$ if $(p, q) \in R^+$ (where R^+ is the transitive closure of R). From the scheduling point of view, we are interested in finding the variables that will have an effect on scheduling, also outside the actor in which the variable exist. For this, the first step is to find the input ports used by guards, $P_{ig} \subseteq P_i$. The next step is to find the output port which is connected to this input port, and add $\{y \in V_G | (x, y) \in R^+ \wedge x \in P_{og}\}$ to the variables relevant for scheduling $V_S \subseteq V$. If we find that any of these variables corresponds to an input port, this input port is also regarded as a guarded port and added to P_{ig}. Finally, when every guarded input port has been analyzed, the set of variables needed for scheduling is known.

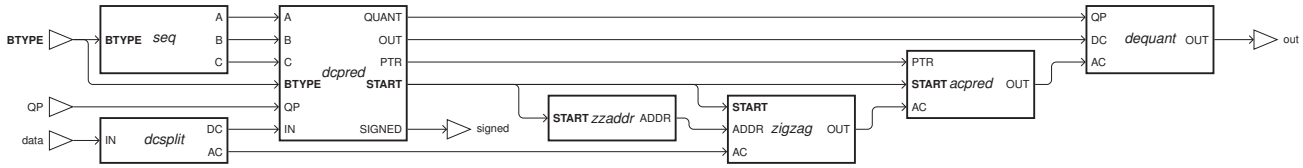

Fig. 1. The acdc network of a RVC-CAL MPEG-4 decoder.

As an example, consider the sub-network in Figure 1. The input ports named *BTYPE* and *START* are the ports which are used in guards (belongs to P_{ig}) in their corresponding actors. The only output port connected to any of these is the *START* output of *dcpred*, for which we then check R^+ for dependencies. As a result, we find the variables in *dcpred* used to generate the *START* output, we also find (in this specific network) that *START* depends on *BTYPE*.

The next step is to use a model checker to find paths between known states of the program. The output from this analysis gives us the variables that must be described by these states in order to generate correct schedules. It also gives us information about the potential complexity of the variables, i.e. the number of possible states of those variables. For example, variable $x \in V_S$ has a cyclic dependency if $(x, x) \in R^+$. This means that the variable depends on itself and typically is a counter. If the variable also depend on any input port, $\exists y\,(y \in P_i \wedge (x, y) \in R^+)$, the variable is likely to have many states, which might indicate that the guard depending on this variable should be resolved at run-time.

4. SCHEDULING WITH A MODEL CHECKER

Model checkers are typically used to verify that a system meets its specifications by defining particular properties that should hold for the system. The model checker searches the state space, and if it finds that one of the specified properties does not hold, it will provide a counterexample, i.e. a state that violates the property.

In the model checker generated from a CAL program, a state is represented by the state variables in V_S, the FSM states and the tokens on the queues in the network; a transition from one state to another corresponds to firing an action in the CAL program. We can therefore initialize the model checker to a specific state and check if a second state is reachable from the first state; if the second state is reachable, then, there exists a sequence of action firings that will result in the second state for the CAL program. In other words, there is a deterministic schedule between the two states.

Finding a sequence of transitions between two specific states in the model checker generated from the CAL program, involves checking conditions representing the guards of the actions. The obtained sequence, however, is a deterministic schedule which is always valid when the network is in the specific state and the correct input is available. For this reason the scheduler must keep track of the state of the network in order to choose a valid schedule. As a consequence, the scheduler often becomes a state machine, which for a specific input chooses different schedules depending on the current state of the network. The current state is in the model checker described by the current value of the variables in V_S, the current state of the actor FSMs, and the state of the FIFOs connecting the actors. In the generated model, the only missing information is the input to the network. In this paper we will consider the set of possible inputs to be known and concentrate on scheduling these correctly.

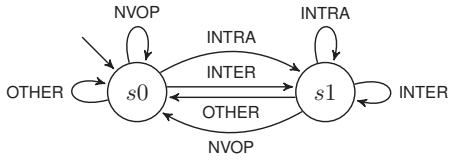

Fig. 2. Scheduler finite state machine

The actual scheduling problem is to find a known state from the current state consuming the given input. To begin with, the only known state is the initial state, while new states are added when no known state can be reached. The goal is to generate a scheduler that can be described as an FSM, such as the example scheduler in Figure 2. This particular scheduler, which was generated for the network in Figure 1, accepts four different input block types and has two states, making it eight schedules. The two states of the scheduler corresponds to the initial state of the network and to one other state where either some variables, FSM states or FIFOs does not have the same value as in the initial state.

For every new state of the scheduler, the model is set to the corresponding network state and each possible input is placed on the input queues of the model to check if any known state is reachable from the current state. To make the model checker search for a specific state, the state is added as a specification of the model to be checked. One possibility is to describe this specification using linear temporal logic (ltl), and we should note that we actually should state that this property should never hold as model checkers are designed to search for unwanted behavior. We can, for example, use the *always* operator to specify that every state should be such that if the input has been consumed ($I0$), the state is not the initial state $S0$, this would be expressed as $\Box\neg(S0 \wedge I0)$.

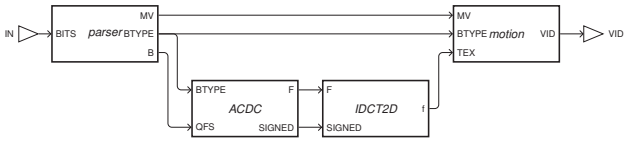

Fig. 3. The MPEG-4 Simple Profile decoder

If no known state is reachable, a new state needs to be introduced. The new state can be any reachable state where the inputs has been consumed, one possibility is to run the model checker in simulation mode to get a hint of how the network behaves and based on this information describe the new state.

5. RESULTS

In this case study we experiment the quasi-static scheduling technique of an MPEG-4 Simple Profile decoder. Figure 3 illustrates the top-level view of the decoder. The quasi-static scheduling is applied separately on the acdc sub-network, the idct2d sub-network and the motion sub-network. The model checker outputs 8 schedules for acdc, 1 schedule for idct and 13 schedules for motion.

Table 1 shows the number of checks (conditional statements) that are executed at run-time on the whole sequence. In this experiment, there is one check when an action is tested for execution (availability of input tokens and guards). The total number of checks is divided by 8 when using the quasi-static scheduling.

# of checks	Scheduling	
	Dynamic	Quasi-static
acdc only	129.4×10^6	0.8×10^6
idct only	189.4×10^6	0.1×10^6
motion only	9.4×10^6	0.9×10^6
parser only	53.1×10^6	51.9×10^6
total	381.3×10^6	53.7×10^6

Table 1. Number of checks for the MPEG-4 SP decoder.

The second experiment consists in comparing the performance of the decoder with a dynamic scheduler and with a quasi-static scheduler for the acdc and idct and motion sub-networks. Table 2 shows the run-time performance of the decoder with and without the static schedules from the model checker. The speed-up of the overall execution is about 29% by using the quasi-static scheduling technique compared to the dynamic version.

scheduling	Dynamic	Quasi-static
frame rate (fps)	91	118

Table 2. Frame rate of the MPEG-4 SP decoder

6. CONCLUSION

This paper presented a method for quasi-static scheduling of dynamic dataflow programs using a state-based scheduling technique. Experiments show that the quasi-static scheduling technique helps to reduce the run-time overhead of the target application and consequently increase its run-time performance. This method used in conjunction with other compile-time scheduling techniques can significantly improve the performance of dynamic dataflow programs, closing up the gap with existing optimized software implementations.

7. REFERENCES

[1] Jani Boutellier, Christophe Lucarz, Sébastien Lafond, Victor Gomez, and Marco Mattavelli, "Quasi-static scheduling of cal actor networks for reconfigurable video coding," *Journal of Signal Processing Systems*, 2009.

[2] R. Gu, J. Janneck, M. Raulet, and S. Bhattacharyya, "Exploiting statically schedulable regions in dataflow programs," *Journal of Signal Processing Systems*, vol. 63, pp. 129–142, 2011.

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

[4] Gerard Holzmann, *Spin model checker, the: primer and reference manual*, Addison-Wesley Professional, first edition, 2003.

[5] ISO/IEC 23001-4:2009, "Information technology - MPEG systems technologies - Part 4: Codec configuration representation," 2009.

[6] Marco Mattavelli, Ihab Amer, and Mickael Raulet, "The reconfigurable video coding standard," *IEEE Signal Processing Magazine*, vol. 27, no. 3, pp. 157–167, 2010.

[7] Matthieu Wipliez and Mickal Raulet, "Classification and transformation of dynamic dataflow programs," in *DASIP '10*, 2010.

[8] Johan Ersfolk, Ghislain Roquier, Fareed Jokhio, Johan Lilius, and Marco Mattavelli, "Scheduling of dynamic dataflow programs with model checking," in *IEEE International Workshop on Signal Processing Systems (SiPS)*, 2011.

[9] Jani Boutellier, Olli Silvén, and Mickaël Raulet, "Scheduling of cal actor networks based on dynamic code analysis," in *ICASSP*, 2011, pp. 1609–1612.

[10] Matthieu Wipliez, Ghislain Roquier, and Jean-François Nezan, "Software code generation for the rvc-cal language," *Journal of Signal Processing Systems*, 2009.