Dataflow programs analysis and optimization using model predictive control techniques: an example of bounded buffer scheduling

M. Canale
Dipartimento di Automatica
e e Informatica, Politecnico di Torino
Italy
S. Casale-Brunet, E. Bezati, M. Mattavelli
EPFL SCI-STI-MM
École Polytechnique Fédérale de Lausanne
Switzerland
J.W. Janneck
Department of Computer Science
Lund University
Sweden

Abstract—This paper presents a new approach to buffer dimensioning for dynamic dataflow implementations. A novel transformation applied to the execution trace graph of a dataflow program is introduced in order to generate an event driven system. It is shown how model predictive control theory techniques can be applied to such a system to analyse the execution space of a dataflow program and to define and to minimize a bounded buffer size configuration that corresponds to a deadlock free execution. Some experimental results obtained using two design examples, i.e. a JPEG and an MPEG HEVC decoder, are reported and compared to the state of the art results in order to show the effectiveness of the introduced approach.

I. INTRODUCTION

In several signal processing applications areas, the use of dataflow programs as a mean to describe the algorithm implementation makes up an interesting alternative to the classical sequential programming approach. In particular, it allows a more extensive program execution analysis which paves the way to a rich variety of parallel implementation solutions, see in this regard [1], [2], [3], [4]. The main reasons of these attractive features rely on the fact that dataflow programs are highly analyzable, platform independent and explicitly expose the potential parallelism of the application. Several dataflow computation models are structured as (hierarchical) networks of communicating computational kernels, called actors. As depicted in Fig. 1, actors are connected by directed, lossless, order preserving point-to-point communication channels, called buffers. Therefore, the flow of data between actors in a dataflow network is fully explicit and data sharing is only allowed by sending data packets, called tokens. This work considers a very general dynamic dataflow Model of Computation (MoC) called Dataflow Process Network (DPN) with firing. A specific property of this MoC is that an actor execution is performed as a sequence of discrete steps (also called firings). During each step an actor can consume a finite number of input tokens, produce a finite number of output tokens, and modify its own internal state if it has any (i.e. defined as state variables and input tokens values). The algorithmic part of a single actor firing is specified inside the so called actions: at each step, according to its state variables and input tokens value, only one action can be fired. The resulting absence of race conditions makes the behaviour of a dataflow program more robust to different execution policies, whether those be truly parallel or imply some interleaving of the individual actor executions.

The two most widely used techniques for the optimization of dataflow program executions are the Model Checking [5], [6], [7], [8], [9] and the Execution Trace Graph (ETG) analysis [10], [11]. The model checking approach is mainly based on an abstract simulation of the design and includes several methods for the dynamic analysis. For example, symbolic representations (e.g. Binary Decision Diagrams) and temporal logic (e.g. Linear Time Logic, Computational Tree Logic) mainly focus on answering yes/no questions that turn performance evaluation into a tedious process. The main disadvantage of the model checking approach is in the fact that often there is no way for searching trade-offs between accuracy and scalability. In fact, when dealing with high dimensional applications, the design space exploration can become easily unfeasible to explore (e.g. also unreachable states are considered) and the designer is left without a partial estimation of the system performance. On the other hand, the methods based on the ETG typically deal with the analysis of a graph data structure, which is obtained after an high level and architecture independent simulation. Therefore, the entire computation can be seen as a collection of nodes that represent the execution steps and a collection of directed arcs that represent the functional dependencies. In this case, analysis methods are based on the processing of a directed acyclic graph. As a consequence, the design space can be efficiently explored using appropriate heuristics and reduced according to the optimization needs without impacting the accuracy of results.

The novelty introduced in this work is represented by the transformation of the ETG into a discrete event system, i.e. a Petri Net (PN). Such a transformation leads to the possibility of exploring the design space by using some of the well known system control strategies in order to optimize the mapping and partitioning of a dataflow application. In particular, it is clarified how such transformation can be efficiently applied also to complex designs and very large execution trace graphs. The evaluation of a bounded memory and deadlock free buffer size configuration of a dataflow program is used as context for showing the powerfulness of this approach, since it is one of the classical and most challenging problem in the domain of dataflow applications. Yet, in general, overall buffer minimization/dimensioning remains an important optimization objective for achieving cost effective implementations on embedded processing systems that may suffer from severe
memory limitations. Obviously, such problem has shown to be NP-complete [5].

The paper is organized as follows: Section II reports a brief overview of the ETG analysis and the PN. Based on those concepts, the new transformation from the execution trace graph to an event based system model is introduced. The transformation is then used in Section IV where the Model Predictive Control (MPC) approach is used to evaluate a bounded buffer size configuration without deadlocks. For readers not familiar with MPC, Section III summarizes the main features of this control approach. Experimental results obtained with two typical signal processing applications, i.e. a JPEG and an MPEG HEVC decoder, are reported in Section V. Finally, conclusions and further perspectives are discussed in Section VI.

Fig. 1: Structural view of a dataflow program.

II. EXECUTION TRACE GRAPH AND PETRI NETS

A summary of the basic concepts of Execution Trace Graph (ETG) and Petri Nets (PN) is reported along this section by means of the example depicted in Fig. 2. In particular, the dataflow program reported in Fig. 2a is composed by three actors (i.e. *Producer*, *PingPong* and *Consumer*) and by two interconnecting buffers (i.e. *b1* and *b2*). At each execution step, as described through the CAL language code definition of the actors reported in Fig. 3, actor *Producer* produces two tokens on *b1*, actor *PingPong* consumes one token from *b1* and produces one token on *b2*, finally actor *Consumer* consumes two tokens from *b2*. It must be noted that actor *Producer* can be executed only twice (i.e. due to the guard on its action *produce2*), consequently for this example the number of overall execution steps is limited. This example is based on a Static Dataflow Network (SDF) [5], however the proposed methodology can be successfully applied to more expressive MoCs (e.g. in Section V this approach is used for two dynamic dataflow programs).

A. Execution Trace Graph

As discussed in Section I, the execution of a dataflow program belonging to the general class of DPN with firings, can be seen as a collection of steps. Each step can consume a finite number of input tokens, produce a finite number of output tokens, and modify its own internal state if any. Consequently, it is possible to characterize the intrinsic dependencies among two steps. For example, if a step consumes tokens, it must rely on the execution of the actor that produces such tokens. The same can be stated if one step makes use of a state variable that was previously modified by another step. Different kinds of dependencies can be defined and used to characterize the execution of a dataflow program (i.e. Finite State Machine, State Variable, Guard, Port and Tokens dependencies [11]). Defining dependencies among executed steps establishes a precedence order: if step $s_2$ depends on step $s_1$, then $s_1$ has to be executed and completed before $s_2$ can be started. The transitive hull of the dependencies is the precedence relation $\prec$, so the precedence constraint between $s_1$ and $s_2$ can be expressed as $s_1 \prec s_2$. The set of steps $\mathcal{S}$ and dependencies $\mathcal{D}$ can be represented using a directed and acyclic graph $G(\mathcal{S}, \mathcal{D})$ called ETG, where each single step is represented by a node and each dependence by a directed arc. For the sake of this work, it is possible to group the different dependency kinds into two groups:

- the *internal dependencies* (i.e. Finite State Machine, State Variable, Guard and Port) that describe the relations between two steps of the same actor;
- the *tokens dependencies* that describe the relation between two steps that respectively produce and consume at least one token.

Considering the dataflow program reported in Fig. 2, the ETG corresponding to its execution is depicted in Fig. 2b. In this case, the *steps* set is defined by $\mathcal{S} = \{s_1, s_2, \ldots, s_8\}$ where the step identifiers (i.e. actor, action name and execution index) are reported in Table I. Dependencies $\mathcal{D}$ are represented by directed arcs: tokens dependencies by blue arrows, and internal dependencies by green arrows. In the reported example, $s_4$ depends on $s_3$ since both belong to the same actor *PingPong* (i.e. $s_3 \prec s_4$), moreover both depend on $s_1$ since they consume tokens produced by this latter (i.e. $s_1 \prec s_3 \prec s_4$).

In the domain of dynamic dataflow programs, the ETG size (i.e. the number of steps and dependencies) depends in general by the execution input stimulus: for systems implementing signal processing applications (e.g. video or audio codecs) sets of input stimuli that are sufficiently complete to statistically cover the whole span of the application behavior are needed to generate statistically meaningful ETGs. Thus, one challenge becomes how efficiently handle such possibly very large ETGs. In this work, as discussed in Section IV, this problem is solved finding suitable trace-cuts.

B. Petri Nets

A PN is a particular kind of bipartite directed graphs made up by three types of objects: places, transitions, and directed arcs [12]. Directed arcs connect places to transitions or transitions to places. Each place can contain tokens: the presence or the absence of a token can indicate whether a condition associated with this place is true or false. Formally, a PN is defined as $\mathbb{N}(P, T, I, O, M_0)$, where: $P = \{p_1, p_2, \ldots, p_m\}$ is a finite set of places, $T = \{t_1, t_2, \ldots, t_n\}$ is a finite set of transitions, $I : P \times T \subseteq \mathbb{N}$ is the pre-incidence matrix that defines directed arcs from places to transitions, $O : T \times P \subseteq \mathbb{N}$ is post-incidence matrix that defines directed arcs from transitions to places, $M_0 \subseteq \mathbb{N}$ is the initial marking of places. The execution of a PN is controlled by the number and distribution of tokens over the places. Similarly to the DPN with firings dataflow model, also a PN executes by firing transitions governed by enabling and firing rules. In a PN a
transition \( t \) can be enabled if all its input places contain at least a number of tokens equal to the weight of the respective directed arc. The firing of an enabled transition removes from each input place the number of tokens equal to the weight of the respective input directed arc and deposits in each output place a number of tokens equal to the weight of the respective directed output arc. Mathematically, firing the transition \( t \) at event \( k \) yields to a new marking:

\[
M(p, k + 1) = M(p, k) - I(t, p, k) + O(t, p, k)
\]  

(1)

for any \( p \in P \) at each firing instant \( k \in \mathbb{N} \).

C. Event Driven System model from an Execution Trace Graph

Generally speaking, if a dynamic dataflow program belonging to the Dataflow Process Network class is translated into a PN representation, it is required that the MoC is modified accordingly (e.g., using colored PN that allow tokens to carry values and preserves order in their places). However, an alternative way is to transform its ETG. Here the objective becomes to obtain a mathematical formulation of an ETG similar to the one provided by a PN which is synthesized in Equation (1). A straightforward way is to correlate the dependencies constraints defined in ETG with the firing rules of PN. Intuitively, a transition \( t \) can be represented as a step \( s \) that can be fired only if there are enough input tokens on its incoming places \( p \). In other words, tokens availability represents the satisfaction of the respective dependence. Moreover, the number of tokens defined for a particular token dependence can be represented with a corresponding PN arc weight. With such simple rules, the ETG depicted in Fig. 2b can be transformed into a PN as illustrated in Fig. 2c. As can be seen, transitions directly corresponds to steps (i.e. \( S \equiv T \)) and dependencies to places (i.e. \( D \equiv P \)). Additionally, for each step without incoming dependencies the respective transition has an incoming place with an initial token (e.g. see \( t_1 \) and \( P_1 \) in Fig. 2c). It is easy to show that the evaluated PN is terminated and matrices \( I \) and \( O \) in Equation (1) are fixed over the entire network firing (i.e. \( I(t, p, k) \equiv I(t, p) \) and \( O(t, p, k) \equiv O(t, p) \)). Consequently, Equation (1) can be formulated as:

\[
\begin{align*}
\left\{ \begin{array}{l}
x(k + 1) = x(k) + B u(k) \\
y(k) = C x(k)
\end{array} \right.
\]

(2)

This expression can also be considered as the equation modelling an event driven system, for which \( x(k) \in \mathbb{N}^m \) represents the number of tokens in each position \( p \) such as \( x(k) = \{M(p, k) : \forall p \in P\} \), \( u(k) \in \mathbb{N}^n \) represents the fired transition at firing \( k \) such as \( u(k) = \{\delta_i : \forall i \in T\} \) (where \( \delta_i = 1 \) if \( t \) has been fired, \( \delta_i = 0 \) otherwise). \( B \in \mathbb{Z}^{m,n} \) represents the number of produced/consumed tokens for each place \( p \) when \( t \) is fired (i.e., the incidence matrix of the PN). \( y(k) \in \mathbb{N}^y \) represents the number of real tokens available in each buffer of the original dataflow network. Considering the example of Fig. 2, matrices \( B \) and \( C \) of Equation (2) are defined respectively as:

\[
B = \begin{bmatrix}
-1 0 0 0 0 0 0 0 \\
1 1 0 0 0 0 0 0 \\
1 0 -1 0 0 0 0 0 \\
0 0 1 -1 0 0 0 0 \\
0 1 0 0 0 0 0 0 \\
0 0 0 0 1 -1 0 0 \\
0 1 0 0 0 0 -1 0 \\
0 0 1 1 0 0 -2 0 \\
0 0 0 0 0 0 1 -1 \\
0 0 0 0 0 0 1 0
\end{bmatrix}
\]

\[
C = \begin{bmatrix}
0 0 1 0 1 0 1 0 1 0 0 \\
0 0 0 0 0 0 0 0 0 1 0
\end{bmatrix}
\]

<table>
<thead>
<tr>
<th>Step</th>
<th>Actor</th>
<th>Action</th>
<th>Action Execution Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Producer</td>
<td>produce2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>PingPong</td>
<td>copy</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>consumer</td>
<td>consume2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>PingPong</td>
<td>invert</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>producer</td>
<td>copy</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>PingPong</td>
<td>invert</td>
<td>2</td>
</tr>
</tbody>
</table>

TABLE I: Execution steps of the ETG depicted in Fig. 2b: step identifier, actor name, action name and action execution index.

III. MODEL PREDICTIVE CONTROL

The key aspects of the Model Predictive Control (MPC) used along the next sections are presented herein below. However, for a more complete presentation the interested reader can refer to [13], [14] where also different domains of application are illustrated. MPC, or known also as receding horizon control, is a model based form of control in which the control action is obtained by solving, at each sampling time, a finite-horizon open loop optimal control problem. Using the current state of the plant as the initial state of the problem, the optimization solution yields to an optimal control sequence. The control loop is then closed by using the first control move obtained from the optimized sequence. This is the main difference from conventional control strategies (e.g. Proportional Derivative Integrator, Linear-Quadratic Regulator) which use a pre-computed control law.

Moreover, one of the main features of MPC is the possibility to take into account hard system constraints directly in the optimization problem. The system evolution is predicted over \( H_p \) (Prediction Horizon) events \( k \). During event \( k \), using the standard notation of discrete event systems (e.g. see Equation (2)), the actual output is represented as \( y(k) \), the predicted output and optimized output for the event \( k+i \) are represented respectively as \( y(k+i|k) \) and \( u(k+i|k) \). At each event \( k \), the MPC strategy calculates a set of \( H_c \leq H_p \) (Control Horizon) values of the input \( U(k)_i^{H_c} = \{u(k+i|k), \forall i \in \{0, 1, \ldots, H_c-1\}\} \). The input are evaluated so that that the predicted outputs \( Y(k)_i^{H_c} = y(k+i|k), \forall i \in \{1, 2, \ldots, H_p\} \) reach the target point in a optimal manner. \( U(k)_i^{H_c} \) is obtained by optimizing a linear or quadratic constrained objective function such as:

\[
J(k) = f(x(k|k),u(k|k),u(k+1|k),\ldots,u(k+H_c-1|k))
\]

(3)
It must be noted that during the prediction the control is receding horizon approach: after evaluating the optimal input and its corresponding Petri net representation.

Fig. 2: Dataflow program network, the execution trace graph and its corresponding Petri net representation.

In other words, at each event $k$ the objective is to minimize an objective function subject to additional constraints as:

$$\min_{u(k), u(k+1), \ldots, u(k+H_c-1)} J(k)$$

subject to

$$\begin{align*}
&y_{min} \leq y(k+i) \leq y_{max}, \forall i \in \{1, 2, \ldots, H_p\} \\
u_{min} \leq u(k+i) \leq u_{max}, \forall i \in \{0, 2, \ldots, H_c-1\} \\
u(k+i) = 0, \forall i \in \{H_c, H_c+1, \ldots, H_p\} \\
g(u(k), u(k+1), \ldots, u(k+H_c-1)) \leq 0
\end{align*}$$

It must be noted that during the prediction the control is held constants after $H_c$ control moves (i.e. $u(k+i) = 0$). As mentioned before, a remarkable feature of MPC is its receding horizon approach: after evaluating the optimal input set $U(k)_{H_c}$ only the first move $u(k)^* = u(k|k)$ is actually implemented. Then, a new sequence is calculated at the next event and again only the first input move is implemented.

IV. Finding a Minimum Buffer Size Configuration

A key issue of dynamic dataflow programs is how to bound and minimize the buffer size configuration without impacting the performance and guaranteeing at the same time a deadlock free execution [5], [6], [7], [10], [11]. This problem has been proved to be a NP-complete problem [5] which needs the use of heuristic algorithms. In this section this problem is solved using the ETG transformation and treating the program like a linear discrete event system as illustrated in Section II. Considering an ETG with $H_s = |S|$ steps, the problem of bounding (and minimizing) the buffer size configuration guarantees at the same time a deadlock free execution can defined such as:

$$\begin{align*}
&\min_{u(k), u(k+1), \ldots, u(k+H_s)} J = \sum_{k=1}^{H_s} \sum_{j} y(k) \\
&\text{subject to} \\
&y(k)_j \geq 0, \forall k \in \{1, 2, \ldots, H_s\}, \forall j \in \{1, 2, \ldots, b\} \\
&\sum_{k=1}^{H_s} w(k)_j = 1, \forall k \in \{1, 2, \ldots, H_s\} \\
&\sum_{k=1}^{H_s} w(k)_i = 1, \forall i \in \{1, 2, \ldots, H_s\}
\end{align*}$$

where $y(k)_j$ represents the $j$-th component of $y(k)$, i.e. the number of tokens available on the $j$-th buffer, the constraint $\sum_{k=1}^{H_s} w(k)_i = 1, \forall k \in \{1, 2, \ldots, H_s\}$ requires that a step be fired at each event $k$, while the constraint $\sum_{k=1}^{H_s} w(k)_i = 1, \forall i \in \{1, 2, \ldots, H_s\}$ requires that a step must be fired only once (i.e. the deadlock condition is avoided because all the steps in $S$ must be fired). In other words, firing only one step at each event $k$, can also be seen as finding a topological order of $S$ for which the sum of all the available tokens along the
It is worth remarking that the ETG is a directed acyclic graph containing all the steps fired during the execution of a dataflow program and it is completely independent from any buffer size configuration [11]. Hence, an ETG can have different topological orders (i.e. the ordered sequence of the fired step transitions) with different minimal buffer size requirement for admitting a deadlock free execution. Therefore, the initial problem can be relaxed: sorting at each event (fired step transitions) with different minimal buffer size requirement for admitting a deadlock free execution. Consequently, this problem can be reduced to the one defined in (4) where, in this case, $H_p$ (i.e. the prediction horizon) defines the steps number of each trace-cut. At each event $k$ the trace cut $S(k)_{H_p} \subseteq S$ is evaluated so that it contains only $H_p$ unscheduled steps of $S$ with the lowest available topological order. Then, according to the procedure just described, the optimization problem can be formulated as:

\[
\text{minimize} \quad J(k) = \sum_{i=1}^{H_c} \sum_{j=1}^{H_p} y(k+i|k)j \\
\text{subject to} \quad y(k+j|i|k) > 0, \forall j, i \in \{1, 2, \ldots H_p\} \\
\sum_{i} u(k+j|i|k) = 1, \forall j \in \{0, 1, 2, \ldots H_c - 1\} \\
\sum_{i} u(k+j+i|k) = 0, \forall j \in \{H_c, H_c + 1, \ldots H_p\}
\]

where $H_c$ (i.e. the control horizon) is the number of steps that can be fired (i.e. ordered) inside $S(k)_{H_p}$. At each event $k$ only the first selected step defined by $u(k)^* = u(k|k)$ is fired. When firing the step defined by $u(k)^*$ the number of tokens inside each buffer is updated accordingly. Only when all the steps have been fired the minimal buffer size configuration can be defined such as the maximal token capacity of each buffer obtained during the entire execution.

**V. EXPERIMENTAL RESULTS**

In order to evaluate the effectiveness of the proposed approach a JPEG and an HEVC decoder, both conforming to the MPEG RMC standard (ISO/IEC 23001-4:2011) [15], [16], [17], [18], have been implemented using the RVC CAL dataflow language [19]. The high level structure of these designs are depicted in Fig. 4. The experiments have been performed using the Orc-Apps RVC-CAL designs [20], while the ETGs and the implementation of the algorithms have been made using the TURNUS framework [21], [22] developed by the authors. The number of actors and buffers of each design is reported on Table II together with the number of steps contained in each ETG used for the analyses. Tables III and IV report the results obtained with our approach respectively for the JPEG and the HEVC decoder. For each configuration of $H_p$ it is also reported the ratio with the number of actors defined by $\lambda(H_p)$. Results have been also compared with the one obtained using [23]: where $\Delta\text{bits}_{\text{ETG}}$ and $\Delta\text{tokens}_{\text{ETG}}$ represents the difference obtained with the new approach respectively in terms of bits and tokens. Moreover, for each configuration the average time required for solving at each iteration the problem proposed on Equation (6) is reported in terms of ms.

Results evaluation has been obtained using a desktop PC with an i7 CPU and 32GB of memory. It can be observed that even with very small trace-cuts such that $\lambda(H_p) = 1$ and a single optimized fired step such as $H_c = 1$, the use of the proposed approach leads to a bounded buffer size configuration that is around 15% smaller compared with the well established solution introduced in [23] (i.e. both in terms of tokens and bits). It must be noted that if a buffer can contain only tokens of the same type (e.g. unsigned/signed integer, floats) where the number of bit for a single token is known (as in the case of the two proposed examples) then the optimization objective $J(k)$ of (6) can be easily formulated in terms of tokens by introducing a cost value for each component of the vector $y$. 
In this work, a novel method for representing and optimizing the execution of a dynamic dataflow program as a controlled event driven system is presented. It is shown how a design optimization problem can be expressed in terms of a feedback controlled system, using concepts from both graph and control theory. In particular, the bounded buffer scheduling problem has been reformulated as a control problem by transforming the dataflow execution trace into a Petri network. Experimental results have shown how it is possible to define a bounded and deadlock free buffer size configuration for each execution trace. Moreover, for the proposed design cases, results show that the overall buffer size is about 20% smaller compared to other state of the art techniques. Future works will be focused on using this approach for other design space optimization problems such as mapping and partitioning of a dataflow network over the processing elements of heterogeneous platforms.

ACKNOWLEDGMENT

This work is supported by the Fonds National Suisse pour la Recherche Scientifique, under grant 200021.129960 and grant 200021.138214, and by the strategic research area ELLIIT.

REFERENCES