Cyclo-Static Dataflow: Model and Implementation

Marc Engels, Greet Bilsen, Rudy Lauwereins, Jean Peperstraete,

K.U. Leuven, Department E.S.A.T., Kardinaal Mercierlaan, 94, B-3001 Leuven, Belgium.

Abstract.

Cyclo-static dataflow (CSDF) is used for specifying digital signal processing algorithms with a cyclically changing, but predefined, behavior. Unlike other models for such applications, CSDF allows for static scheduling and hence a very efficient implementation. In this paper we review the CSDF paradigm and present the scheduling approach for CSDF graphs that is currently implemented in the Graphical Rapid Prototyping Environment GRAPE.

Index-terms: Cyclo-static dataflow, digital signal processing, hard real-time systems, multiprocessing, parallel processing, rapid prototyping, static scheduling, synchronous dataflow.

1. Introduction

Digital signal processing (DSP) is used for speech synthesis and recognition, telecommunications, image and video processing, robotics, etc. Nowadays, most new DSP applications are first analysed and simulated on workstations or super-minicomputers and next implemented on application-specific integrated circuits (ASICs) or dedicated hardware. Prototypes are worked out only during the last design stage because designing dedicated prototyping hardware is time-consuming and costly. However, with a prototype in the earlier stages of the design we could

- verify the application with real-life input signals and true interfaces to the environment (In [1] it is indicated that 50% of the ASICs do not work in their final system due to interfacing problems);
- optimize variables and evaluate an algorithm's subjective qualities under real-time conditions, e.g. by listening to the actual results of an audio algorithm or looking at the output of a video algorithm;
- convince management, marketing people or potential customers of the usefulness of the proposed DSP system;

For the construction of such an early prototype we propose a *general-purpose multi-processor set-up*. To help the designer with mapping his application on the parallel hardware, we advocate *integrated programming environments* like the commercially available DSPStation from Mentor Graphics, Signal Processing Workstation from the Alta Group of Cadence, and COSSAP from Synopsys.

All three environments use variants of dataflow semantics for describing DSP applications. *Dataflow graphs* are a natural paradigm for implementation on a multiprocessor because of their modularity and the explicit presence of concurrency. To implement the dataflow graph on parallel hardware, the environment assigns the nodes of the graph to processing devices, routes the data through the multiprocessor network and determines on each device the execution order of the nodes. This can be done either at run-time (*dynamic*) or at compile-time (*static*) [5]. In a dynamic scheduler, a run-time supervisor determines when nodes are ready for execution and schedules them onto processors. This results in a considerable supervising overhead. Static schedulers, on the other hand, determine an execution order at compile-time. The resulting schedule is then executed periodically on the incoming sample-stream with minor run-time overhead (only synchronisation on external data). To allow for static scheduling the environments severely limit the expressiveness of their dataflow model: only applications with an invariant behavior are supported.

GRAPE-II [4], our experimental research tool, uses a more powerful dataflow model that allows for algorithms with a cyclically changing behavior, without sacrifying the advantage of compile-time scheduling. In the next section we present this cyclo-static dataflow (CSDF) model and its relationship with other models. In section 3 we give an example of a CSDF application. A static scheduling method for CSDF is presented in section 4. And finally, we make some concluding remarks in section 5.

2. The Cyclo-Static Dataflow Model.

A dataflow algorithm and its environment are described as a *directed graph*. The *nodes* represent actors

that transform input data streams into output streams. An actor can be atomic or a hierarchically specified subgraph. On the lowest hierarchical level the functionality of the atomic actor is described in a *host language* e.g. Assembly, C.

The *edges* between the nodes represent channels which carry streams of data. An atomic data object is called a *token*. Channels are considered to be *First-In-First-Out (FIFO) queues*. Although the depth of a FIFO is theoretically unlimited, GRAPE-II allows to fix it for algorithmic reasons to a predetermined value. In the latter case, writing to the channel is blocking, otherwise it is non-blocking. The extension to limited buffer lengths is not strictly essential because it is shown in [3] that such a limited buffer can be modelled with a feedback loop. Nevertheless, it makes the graph a lot more readable and is therefore incorporated in the GRAPE-II model.

When an actor is executed it *consumes* a certain number of tokens from its inputs and it *produces* a number of tokens to its outputs. The program execution is data driven in the following way: an actor is executed as soon as its *firing rule* evaluates as "true" and a processing device capable of executing the actor becomes available. A firing rule is a Boolean expression in the number and/or the value of tokens present in the FIFO's.

In GRAPE-II there is one exception to the dataflow behavior described above: *parameters*. A parameter is an input to an actor which can be considered as a single memory location. A value can be read many times, but each write to it overrides the previous value. The use of these parameters is very versatile, for example the description of algorithmic parameters that can be changed at run-time by the user.

Depending on how the consumption and production together with the firing rules are specified, we can divide graphs or subgraphs into different classes: single-rate dataflow, multi-rate dataflow, cyclo-static dataflow and asynchronous dataflow. We shall now briefly describe these dataflow types.

2.1. Single-rate dataflow (SRDF):

In a *single-rate dataflow* graph the consumption and production on each edge is a single token. A node is fireable if there is at least one token on all its incoming edges. For SRDF, a static schedule can easily be constructed by compile-time scheduling tools. The schedule with minimum period contains exactly one instance for each of the tasks and requires only buffers of unitary length.

Many DSP-applications, however, contain tasks that operate at different rates (e.g. interpolating and decimating filters). Because such applications cannot be described using the single-rate dataflow paradigm, the multi-rate dataflow paradigm was introduced.

2.2. Multi-rate dataflow (MRDF):

In *multi-rate dataflow* consumption and production are constant integers that can be computed at compile-time (*manifest*). In the graph they are represented at the corresponding side of the communication edge (Fig. 1). When task v is executed, it produces x_v^u tokens on edge u, where task w consumes y_w^u tokens from the edge each time it is invoked. The firing rule of an actor becomes "true" if and only if each input FIFO contains at least the number of tokens specified by the consumption of that input.

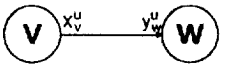

Fig. 1: Multi-Rate Dataflow.

Also multi-rate dataflow graphs can be scheduled at compile-time, leading to an efficient implementation with minimal run-time overhead. The necessary buffer lengths are bounded. The main limitation of the multi-rate dataflow model is that it supposes that every task behaves identically each time it is executed, consuming and producing the same amount of tokens. As a consequence, this does not allow for data-dependent or state-dependent conditionals. In a *data-dependent* node the production or consumption of the node is function of the data-values. In Fig. 2(a), for instance, the node transfers its input token to its upper output if the value is larger than 5 and to its lower output otherwise. In a *state-dependent* conditional the behavior depends on a state-variable of the node. An example is a multiplexer with a predetermined multiplexing pattern, shown in Fig. 2(b). In the first invocation the sample is taken from the upper input and the second time from the lower input. This behavior is then repeated periodically.

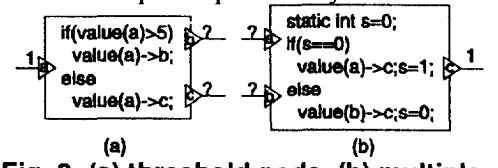

Fig. 2: (a) threshold node, (b) multiplexer.

2.3. Asynchronous dataflow (ADF):

In *asynchronous dataflow* all types of conditionals are supported: the firing rule can be any Boolean expression and consumption and production can be undefined (unknown at compile-time) because they depend on the actual value of tokens. (Often the program can give some

unprecise specification of production and consumption, e.g. minimum and maximum numbers.) Because of the partial knowledge at compile-time, ADF needs a run-time scheduling mechanism to determine when an actor becomes executable. Moreover, the necessary length of the buffers might become unbounded. The great expressive power of ADF thus results in a high run-time overhead. However, the behavior of many state-dependent conditionals is manifest and hence can be implemented more efficiently. This optimization is exploited in the cyclo-static dataflow paradigm.

2.4. Cyclo-static dataflow (CSDF):

In *cyclo-static dataflow* production and consumption are sequences of constant integers (Fig. 3). The production of task v on edge u is $\left[x_v^u(0), x_v^u(1), \ldots, x_v^u\left(P_v^u - 1\right)\right]$, where P_v^u corresponds to the minimum period in the production behavior of task v on edge u. The meaning of these sequences is as follows: the k^{th} time that task v is executed, it produces $x_v^u\left((k-1) \bmod P_v^u\right)$ tokens on the edge. The consumption of task w is completely analogous. The firing rule of a cyclo-static actor evaluates as "true" for its k^{th} firing if and only if all input FIFO's contain at least $x_v^u\left((k-1) \bmod P_v^u\right)$ tokens. Although CSDF is more general than MRDF we will show in section 4 that it is still possible to construct a valid schedule with bounded buffer lengths at compile-time.

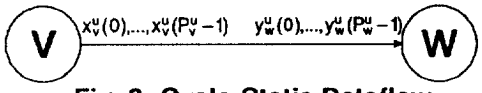

Fig. 3: Cyclo-Static Dataflow.

2.5. Taxonomy:

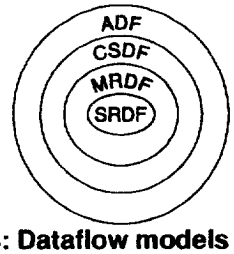

Fig. 4: Dataflow models for DSP.

The relation between the four dataflow models is graphically shown in Fig. 4. Remark that a combination of any of these types can occur in a single graph. The GRAPE-II tools automatically detect which subgraphs correspond to a given dataflow type. As a consequence, they can take actions optimally suited for each part of the application, without a need for the programmer to specify whether it resides in a specific dataflow domain.

3. Cyclo-static dataflow example.

In this section, we illustrate the advantages of CSDF with a system for evaluating codebooks for a Code Excited Linear Prediction (CELP) algorithm. In this system, an incoming speech signal is filtered by a Lineair Prediction Coding (LPC) analysis filter to produce a noise-like residual. After determining the filter-coefficients on a frame of 80 speech samples, these samples are filtered one by one. Blocks of 40 filter outputs are compared with a codebook of 1024 reference blocks. The number of the best-matching codebook vector is send serially through the channel. At the receiving side, the lookup element takes the appropriate vector out of the codebook and sends it to the LPC-synthesis filter. This filter reconstructs the speech signal out of the codebook vector and the corresponding filter-coefficients which are also transmitted to the receiver.

In a straight-forward multi-rate specification of this algorithm, 80 invocations of the LPC-analysis and LPC-synthesis are grouped together in one task (Fig. 5a). The corresponding schedule with maximum parallelism is shown in fig. 5b.

Definition - The *maximally parallel schedule* of a dataflow application is a schedule with in-order execution of the tasks, on an unlimited number of devices and where all communication is supposed to take zero time.

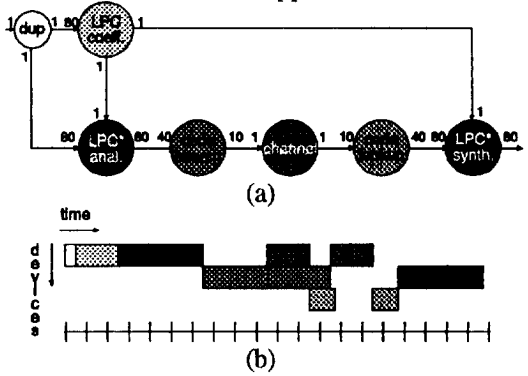

fig. 5: (a) MRDF specification of the CELP-algorithm; (b) maximally parallel schedule.

The CSDF specification of the same algorithm (Fig. 6a) is more natural because it does not require the grouping of 80 filter tasks. As a consequence, the schedule has a smaller granularity and a higher inherent parallelism. This is clearly shown in its maximally parallel schedule (Fig. 6b).

The higher degree of inherent parallellism in CSDF specifications will in general result in a higher throughput and shorter delays. The latter advantage is essential in systems with a feedback loop and it also reduces the amount of buffer memory.

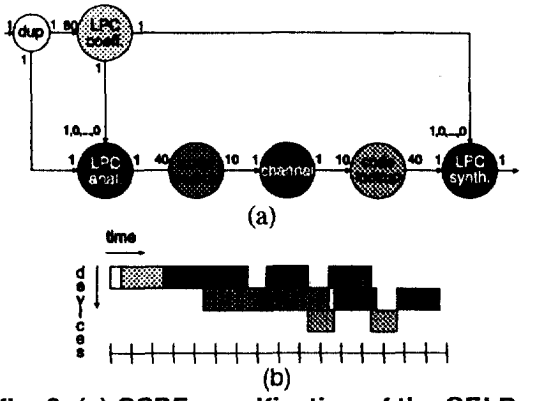

fig. 6: (a) CSDF specification of the CELP-algorithm; (b) maximally parallel schedule.

4. Scheduling of CSDF graphs.

An essential step in the implementation of a cyclo-static dataflow graph on a *control-flow* multiprocessor is the static scheduling of the actors. The resulting schedule will be repetitively executed on the processors. For a proper run-time execution it must guarantee that all necessary data is available when a task is executed and that the amount of data in the buffers remains non-negative and bounded.

The underlying theory for the scheduling of cyclo-static dataflow is an extension of Lee's work on the scheduling of multi-rate dataflow [5]. After an overview of the used notations, we present a summary of these results. In subsection 4.3., we introduce a straight-forward algorithm for obtaining a valid schedule. Afterwards we explain how the scheduling algorithm of GRAPE-II obtains a makespan optimal result.

4.1. Notations.

Following notations are used to describe the production of task v and the consumption of task w on edge u:

- P_v^u, P_w^u — period of the production (consumption) sequence of task $v(w)$ on edge u.
- $x_v^u(k), y_w^u(k)$ — k-th element in the production (consumption) sequence of task v (w) on edge u, with $0 \le k \le P_v^u - 1 (P_w^u - 1)$
- P_v, P_w — least common multiple of $\{P_v^u(P_w^u)\}$, taken among all inputs and outputs of task $v(w)$.
- $X_v^u(i), Y_w^u(i)$ — number of tokens produced (consumed) by task $v(w)$ on edge u during the first i invocations, or $X(Y)_v^u(i) = \sum_{k=0}^{i-1} x(y)_v^u(k)$.

4.2. The period of the schedule.

Our first aim is to obtain the period of the schedule. Next, we will construct an actual schedule with this period. This schedule contains exactly q_v instances for each task v in the graph, where q_v is given by theorem 1.

Theorem 1 - In a cyclo-static graph, q_v is given by:

$$q_v = P_v . r_v$$

where the sequence repetition vector $\vec{r} = [r_1, r_2, \ldots, r_v, \ldots]$ is a solution of the balance equation:

$$\Gamma . \vec{r} = 0 \qquad (1)$$

with:

$$\Gamma_{ij} = \begin{cases} (P_j / P_j^i).X_j^i(P_j^i) & \text{if task } j \text{ produces on edge } i \\ -(P_j / P_j^i).Y_j^i(P_j^i) & \text{if task } j \text{ consumes from edge } i \\ 0 & \text{otherwise} \end{cases} \qquad (2)$$

Proof - The required repeatability of the schedule forces that every node steps through a complete production and consumption sequence. This implies that the number of invocations q_v of a task v must be a multiple of all the periods on its inputs and outputs. As a consequence, q_v must also be a multiple of their least common multiple P_v, or:

$$q_v = P_v . r_v = \left(P_v / P_v^u\right).P_v^u . r_v \qquad (3)$$

To keep the buffer memory bounded and non-negative, the amount of tokens produced on an edge, during a single iteration of the schedule must equal the amount of data consumed from it. For any edge u between tasks v and w this gives:

$$X_v^u(q_v) = Y_w^u(q_w) \qquad (4)$$

For the numbers of tokens produced by v and consumed by w we get:

$$X_v^u(q_v) = X_v^u\left(\left(P_v / P_v^u\right).P_v^u . r_v\right) = r_v.\left(P_v / P_v^u\right).X_v^u(P_v^u) \qquad (5)$$

$$Y_w^u(q_w) = r_w.\left(P_w / P_w^u\right).Y_w^u(P_w^u) \qquad (6)$$

Combining formulas (4), (5) and (6) gives:

$$\left(\left(P_v / P_v^u\right).X_v^u\left(P_v^u\right)\right).r_v - \left(\left(P_w / P_w^u\right).Y_w^u\left(P_w^u\right)\right).r_w = 0 \qquad (7)$$

for every edge u in the graph, leading to the matrix of equation (2). □

Remark that for a connected cyclo-static graph with s nodes, a necessary condition for the existence of q_v is: rank (Γ) = $s - 1$. We conclude this section by deriving a necessary and sufficient condition for the existence of a static schedule for a connected cyclo-static graph. Because of limited space, the proofs of the theorems are ommitted.

Definition - A *single-rate equivalent* of a cyclo-static graph is a single-rate graph with the same scheduling properties as the original graph.

Theorem 2 - For a connected cyclo-static graph with s nodes, rank (Γ) = s-1 is a necessary and sufficient condition for the existence of a single-rate equivalent..

Theorem 3 - A necessary and sufficient condition for the existence of a static schedule for a connected cyclo-static graph, is twofold: a single-rate equivalent must exist and every loop in the single-rate equivalent must contain at least one delay element.

4.3. Constructing a schedule.

Once the scheduling period is determined, we construct a schedule. A straight-forward algorithm to find a valid, but not necessary optimal, schedule starts from the initial state of the graph. It first determines which nodes are fireable. In a cyclo-static graph, an instance of a task is fireable if all previous instances of that task are executed and enough data are present in the input buffers to fire the current instance as well. One of these fireable tasks is then selected at random and used to expand the partial schedule. This procedure is repeated recursively until q_v instances are scheduled for each task v in the graph. The state then equals the initial state and the schedule can be repeated infinitely.

4.4. Scheduling in GRAPE-II.

In GRAPE-II this straight-forward scheduling algorithm is modified to find a makespan optimal schedule. Instead of randomly traversing the search space, we use a directed search procedure, combined with backtracking. Before selecting a task from the list of fireable tasks, the scheduler orders this list, in ascending order of the expected makespan for the final schedule. The partial schedule is then extended with the first element from this ordered list. Once a complete schedule has been constructed a backtracking phase is started. During this backtracking the scheduler checks if there exists a better schedule than the one yet obtained. It goes back to the last decision point where there are still fireable tasks left and selects the next element in the ordered list for schedule expansion. From such point on the forward procedure is resumed until a complete schedule is obtained or a partial one is no longer expected to result in a shorter makespan than the one of the already obtained schedule.

The heuristics used for both the ordering of tasks and the calculation of lower bounds can be found in [2].

5. Conclusions.

In this paper we presented cyclo-static dataflow, a new dataflow paradigm that allows graphs with cyclically changing data-dependencies, without sacrifying the possibility of static scheduling. The importance of a CSDF description for DSP-aplications was illustrated with an example. We also showed how a valid static schedule can be constructed for CSDF applications. The makespan optimal scheduling algorithm of GRAPE-II was briefly introduced. Although GRAPE-II is an environment for DSP prototyping, cyclo-static dataflow and the corresponding scheduling techniques are more generally usable for the specification and implementation of DSP applications, e.g. in simulators and silicon compilers.

Acknowledgement.

Marc Engels is a Senior Research Assistant, Greet Bilsen is a Research Assistant and Rudy Lauwereins is a Senior Research Associate of the Belgian National Fund for Scientific Research. K.U.Leuven-ESAT is a member of the DSP-Valley™ network. This project is partially sponsored by the Belgian Interuniversity Pole of Attraction IUAP-50 and the European Community Esprit-III project 6800 Real-Time DSP Emulation System.

References.

[1] Anonymous, "ASIC Emulation Debugs ASICs And Their Systems", VLSI System Design, Nov. 1988, p. 105.

[2] G. Bilsen, P. Wauters, M. Engels, R. Lauwereins, J.A. Peperstraete, "Development of a static load balancing tool," Proc. of the Fourth Workshop on Parallel and Distributed Processing '93, pp. 179-194, Sofia, Bulgaria, May 4-7, 1993.

[3] J.B. Dennis, G-R Gao, K.W. Todd, "Modeling the Weather with a Data Flow Supercomputer", IEEE Trans. on Computers, Vol. C-33, No. 7, pp. 592-603, July 1984.

[4] R. Lauwereins, M. Engels, M. Adé, J. A. Peperstraete, "GRAPE-II: a System Level Prototyping Environment for Digital Signal Processing Applications", accepted for publication in IEEE Computer.

[5] E.A. Lee, D.G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", IEEE Trans. on Comp., Vol. C-36, No. 1, pp. 24-35, Jan 1987.