# CYCLO-STATIC DATA FLOW.

Greet Bilsen, Marc Engels, Rudy Lauwereins, J.A. Peperstraete

Katholieke Universiteit Leuven, Department ESAT
Kardinaal Mercierlaan 94, B-3001 Heverlee, Belgium
Tel.: +32-16-32 11.11, Fax.: +32-16-32.19.86
email: Greet.Bilsen@esat.kuleuven.ac.be

## ABSTRACT

The high sample-rates involved in many DSP-applications, require the use of static schedulers wherever possible. The construction of static schedules however is classically limited to applications that fit in the synchronous data flow model. In this paper we present cyclo-static data flow as a model to describe applications with a cyclically changing behaviour. We give both a necessary and sufficient condition for the existence of a static schedule for a cyclo-static data flow graph and show how such a schedule can be constructed. The example of a video encoder is used to illustrate the importance of cyclo-static data flow for real-life DSP-systems.

## 1. INTRODUCTION

Multi-processor environments are often used for prototyping and real-time emulation of high-frequency Digital Signal Processing (DSP)-applications to reduce development costs and "time-to-market". To help the designer with mapping the application on the hardware, specialised programming environments were developed (e.g. [1], [2]). Such tools assign the tasks to processing devices, route the data through the network and determine the task-execution order. This can be done either at run-time (*dynamic*) or at compile-time (*static*) [3]. In a dynamic scheduler, a run-time supervisor determines when blocks are ready for execution and schedules them onto processors as these turn idle. Due to the supervising overhead, such schedulers are mostly not suited for DSP-applications that have to operate at very high sample-rates (order of MHz) [4]. Static schedulers, on the other hand, determine a task execution order at compile-time. This schedule is then executed periodically on the incoming sample-stream with minor run-time overhead (only synchronisation on external data is required), such that real-time performance can be obtained.

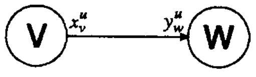

Fig. 1. Synchronous data flow.

The existing mapping tools however can only construct static schedules for applications that fit in the *synchronous data flow* model [1], [3]. In synchronous data flow every task behaves identically each time it fires, producing and consuming the same fixed amount of tokens ($x_v^u$ respectively $y_w^u$ in fig. 1). If for all edges $u$, $x_v^u \equiv y_w^u \equiv 1$, the application is said to be *single-rate*, otherwise it is a *multi-rate* application. In practise many applications do not have a fixed but a cyclically changing behaviour, like the audio example of fig. 2. At the input of the system left and right channel samples arrive alternatively. A demultiplexer consumes the arriving sample and passes it to the appropriate successor block, being the left channel operator for every $(2n+1)$th and the right channel operator for every $2n$th sample. Although this behaviour is known exactly at compile-time, it does not fit in the synchronous data flow model and hence a static schedule cannot be constructed with the classical tools.

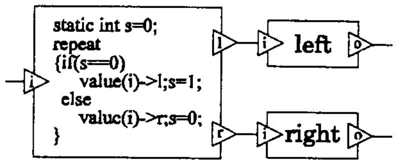

Fig. 2. Left-right audio channel example.

To solve this problem, we introduced the *cyclo-static data flow* paradigm. In cyclo-static data flow the number of tokens produced and consumed by a single task is still known at compile-time, but changes periodically as is shown in fig. 3. In this graph, task $v$ produces $x_v^u(i)$ $(1 \leq i \leq P_v^u)$ tokens every $(n * P_v^u + i)$th time it is invoked. The consumption behaviour of task $w$ is similar.

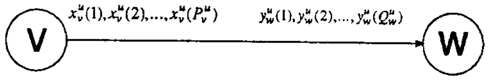
V
$x_v^u(1), x_v^u(2), \ldots, x_v^u(P_v^u)$
$y_w^u(1), y_w^u(2), \ldots, y_w^u(Q_w^u)$
W

Fig. 3. Cyclo-static data flow.

The rest of this paper is constructed as follows. In part 2 we discuss the schedulability of cyclo-static data flow graphs. In part 3 the example of a video encoder is used to accentuate the importance of cyclo-static data flow for real life DSP-applications. And finally in part 4 some conclusions are given.

## 2. CONSTRUCTION OF A STATIC SCHEDULE FOR CYCLO-STATIC APPLICATIONS

In this section we will discuss the schedulability of cyclo-static data flow graphs. After introducing some notations in 2.1., we present both a necessary and sufficient condition for the existence of a static schedule. Two scheduling approaches are discussed in 2.3. and in 2.4. we show how scheduling is actually done in

0-7803-2431-5/95 $4.00 © 1995 IEEE

GRAPE-II. Due to space limitations, the proofs to all theorems had to be dropped.

### 2.1. Notations.

*Global notations*:

| | |
|---|---|
| $N_G$: | number of nodes in a cyclo-static data flow graph $G$ |
| $v(n_v)$: | $n_v$ th invocation of task $v$; i.e. the $n_v$ th time that task $v$ is invoked during the execution of the schedule |
| $n \bmod_1 m$ | $\equiv (n-1) \bmod m + 1$ |
| $n \operatorname{div}_1 m$ | $\equiv (n-1) \operatorname{div} m + 1$ |

*Notations to describe the production of task v and the consumption of task w on edge u:*

| | |
|---|---|
| $ini(u)$: | number of initial tokens on edge $u$ |
| $P_v^u$, $Q_w^u$: | period of the production (consumption) sequence of task $v$ ($w$) on edge $u$ |
| $P_v$: | least common multiple of $\{P_v^u, Q_v^{u'}\}$, taken among all inputs and outputs of task $v$ |
| $x_v^u(k), y_w^u(k)$: | number of tokens produced (consumed) by task $v(w)$ on edge $u$ during its $k$th invocation $= \begin{cases} k^{\text{th}} \text{ element in the production (consumption) sequence of task } v\ (w) \text{ on edge } u; \text{if } 1 \le k \le P_v^u\ \ (Q_w^u) \\ x_v^u(y_w^u) \bmod_1 P_v^u(Q_w^u); \text{if } k > P_v^u(Q_w^u) \end{cases}$ |
| $X_v^u(i), Y_w^u(i)$: | number of tokens produced (consumed) by task $v(w)$ on edge $u$ during the first $i$ invocations, or $X_v^u(i) = \sum_{k=1}^{i} x_v^u(k)\ \left(Y_w^u(i) = \sum_{k=1}^{i} y_w^u(k)\right)$.(For $i$ = 0, we define $X_v^u(0) = Y_w^u(0) = 0$.) |

### 2.2. Conditions for a static schedule to exist

The static schedule of a DSP-application will be executed repetitively on the incoming data-stream. For a proper run-time execution it must guarantee that all necessary data are available when a task is executed and that the amount of data in the buffers remains bounded. This imposes certain conditions on the graph. For a multi-rate graph a necessary condition for the existence of a valid static schedule is presented by Lee in [3]. His theory can easily be extended towards cyclo-static data flow graphs as well. The result of this is given in theorem 1.

*Definition* - Given a connected cyclo-static data flow graph $G$. A vector $\vec{q}_G = [q_1, q_2, \ldots, q_{N_G}]^{\mathrm{T}}$ representing the number of invocations of the tasks of $G$ in a valid static schedule is called a *repetition vector* of $G$.

*Theorem 1* - For a connected cyclo-static data flow graph $G$, a repetition vector $\vec{q}_G = [q_1, \ldots, q_{N_G}]^{\mathrm{T}}$ is given by:

$$\vec{q}_G = \overline{\overline{P}}.\vec{r}, \mathit{with}\, P_{ij} = \begin{cases} P_i; \text{if } i = j \\ 0; \text{otherwise} \end{cases} \tag{1}$$

where the *sequence repetition vector* $\vec{r} = [r_1, r_2, \ldots, r_{N_G}]^{\mathrm{T}}$ is a positive integer solution of the balance equation:

$$\Gamma.\vec{r} = 0 \tag{2}$$

and where the *topology matrix* $\Gamma$ is defined by:

$$\Gamma_{ij} = \begin{cases} (P_j / P_j^i).X_j^i(P_j^i); \text{ if task } j \text{ produces on edge } i \\ -(P_j / Q_j^i).Y_j^i(Q_j^i); \text{ if task } j \text{ consumes from edge } i \\ 0; \text{ otherwise} \end{cases} \tag{3}$$

Theorem 1 shows that a repetition vector and hence a valid static schedule can only exist if equation (2) has a positive integer solution. A graph that meets this requirement is said to be *consistent*. Although consistency is required, it is not sufficient for a valid schedule to exist. Balancing production and consumption on the edges does not exclude deadlocks. If such a deadlock-free schedule actually exists, the graph is said to be *alive*. Sufficient conditions for a consistent multi-rate graph to be alive were given by Lee in [3] and by Karp and Miller in [5]. A method to check the aliveness of cyclo-static data flow graphs is given in theorem 3. Like in the method of Karp and Miller, we only have to check the singular loops in the graph to control its aliveness.

*Definitions* - A *loop L* in a cyclo-static data flow graph is a directed path $v_1 \xrightarrow{u_{1,2}} v_2 \xrightarrow{u_{2,3}} \ldots \xrightarrow{u_{N_L-1,N_L}} v_{N_L} \xrightarrow{u_{N_L,1}} v_1$, where the end-node equals the start-node. A *singular loop* is a loop where every task appears only once. The *first-producing-invocation* $FPI(v_j, n_j, L)$ of $v_j(n_j)$ in $L$, is the first invocation $v_j(s_j)$ $(s_j \ge n_j)$ with non-zero production towards $v_{j+1}$. The *first-dependent-successor-invocation* $FDSI(v_j, n_j, L)$ of $v_j(n_j)$ in $L$ is the first invocation of $v_{j+1}$ that requires data produced by $v_j(n_j)$ to become executable. The *first-data-sequence* $FDS(v_j, n_j, L)$ of $v_j(n_j)$ in $L$, is the invocation sequence $v_j(n_j) \ldots v_j(s_j) v_{j+1}(n_{j+1}) \ldots v_{j+1}(s_{j+1}) \ldots v_{N_L}(s_{N_L}) v_1(n_1) \ldots v_{j-1}(s_{j-1}) v_j(n_j')$, where $s_k = FPI(v_k, n_k, L)$ and $n_k = FDSI(v_{k-1}, s_{k-1}, L)$. (For the last element in the sequence we have $n_j' = FDSI(v_{j-1}, s_{j-1}, L)$).

To construct an $FDS$-sequence, we start from $v_j(n_j)$ and pass by successive invocations of $v_j$, until we find an invocation $s_j$ that produces data towards $v_{j+1}$. Then we pass to $v_{j+1}(n_{j+1})$, where $n_{j+1}$ is the first invocation of $v_{j+1}$ that requires data from $v_j(s_j)$. Here again we pass through a number of subsequent invocations of $v_{j+1}$ and go further towards $v_{j+2}$ once we found a data-producing invocation $v_{j+1}(s_{j+1})$. This procedure is then repeated until we finally arrive at invocation $n_j'$ of task $v_j$. The $FDSI$-calculations involved can be done by use of theorem 2, while the determination of the $FPI$ of a non-producing task-invocation requires a partial traversal of the production sequence.

*Theorem 2* - In a cyclo-static data flow graph, the first token produced during the $n_v$ th invocation of task $v$ on edge $u$ ($v \xrightarrow{u} w$) will be consumed during the $n_w$ th invocation of task $w$, where $n_w$ is given by:

$$\begin{aligned} & n_w = \{[ini(u) + X_v^u(n_v - 1)] \mathit{div}\, Y_w^u(Q_w^u)\} * Q_w^u + n_w^* \\ & \text{with:} \\ & 1 \le n_w^* \le Q_w^u \\ & Y_w^u(n_w^* - 1) \le [ini(u) + X_v^u(n_v - 1)] \mathit{mod}\, Y_w^u(Q_w^u) < Y_w^u(n_w^*) \end{aligned} \tag{4}$$

*Theorem 3* - Given a singular consistent cyclo-static loop $L$ with basic repetition vector $\vec{q}_L$ and $v_j$ an arbitrary node of $L$. $L$ is alive iff $FDS(v_j,s_j^k,L)=v_j(s_j^k)v_{j+1}(n_{j+1}^{k+1})...v_j(n_j^{k+1})$ ends on an instance $n_j^{k+1}>s_j^k$ of $v_j$, for $0\leq k\leq m$. In this formula $s_j^k=FPI(v_j,n_j^k,L)$, $n_j^0$ is the first invocation of $v_j$ that is not initially executable and $m$ corresponds to the first invocation of $v_j$ for which $FDS(v_j,s_j^m,L)$ passes by a task-invocation $n_i^{m+1}>q_i$ of a task $v_i$ of $L$.

By selecting the task with the smallest number of data-producing invocations, we can minimise the calculation work required to check the aliveness of the graph. An upper bound on this calculation work per loop is then proportional with the amount of tasks in it and the minimal number of data producing invocations.

## 2.3. Scheduling approach

Once we have determined a repetition vector and checked the aliveness of the graph, a multiprocessor schedule for the application can be constructed. There are two basic strategies for this.
The first approach consists in transforming the original cyclo-static data flow graph into a single-rate equivalent and construct a schedule by use of existing single-rate tools. A number of disadvantages are associated with this method.

- The size of the graph increases enormously.
- When successive task-invocations are assigned to different processors, internal status is not handled properly. The only way to solve for this problem consists in making the status external prior to the graph transformation and consider it as a normal token that needs to be transported. Checking whether a task contains status and making it external is a hard task, certainly when dealing with library elements. In these elements the internal structure of the task is mostly hidden to the user, such that no information about status is available and that it cannot be made external.
- Another point of attention concerns the management of the data-transport between task-instances. When a task-instance requires data from more than one instance of a predecessor task, a special "collect" task [2] is required to put those tokens in the right order for consumption. Similarly a spread task is required when output samples need to be send to more than one successor instance. When tasks are significantly mixed, these spread and collect tasks become quite complicated to implement and cause extra schedule overhead.

To avoid these problems, we propose a different method, where we work with the original graph. In this approach we force all instances of a task to be executed on the same processing device. The automatic serialisation involved, has a number of advantages compared to the graph-transformation method.

- Correct handling of internal status is intrinsic to the method. A status variable calculated and stored by one instance of a task will automatically be used by the next instance of the task on the same device, always being the next time-instance.
- All task outputs are automatically generated in the right order for consumption by a successor task. This makes that no special data-management tasks (collect and spread) are required anymore, but that FIFO-buffers are sufficient.
- Program as well as data-memory is reduced since code does not need to be duplicated over more devices.
- Another advantage of this method is that the graph itself does not need to be transformed into a single-rate equivalent prior to scheduling. This limits the memory required to store the application and keeps the problem smaller and more easy to handle.

A drawback of this method seems to be a reduction of the parallelism actually exploited, compared to that available in the corresponding single-rate schedule. This may be true when all task-instances are independent. Many tasks in DSP-applications (like filters) however, contain status such that they always need to be executed serially. Even when we succeed in making this status external, the different instances will mostly be assigned to the same device saving a lot of communication overhead. Sometimes however the extra parallelism is required to obtain real-time performance. In this case a (partial) graph-transformation that splits some of the cyclo-static tasks in a number of independent instances, can be performed prior to assignment. Until now such splitting decisions need to be made by the user, but we plan to develop a tool that will automatically detect whether more parallelism is desirable or not.

## 2.4. Scheduling in GRAPE-II

In GRAPE-II assignment, routing and scheduling are performed consecutively in separate tools, each of them working directly on the original graph.
The scheduler uses a depth-first tree-wise search strategy with backtracking to construct a makespan optimal schedule. In this strategy a global schedule is constructed by iteratively extending a partial one with one extra task. In each step all placeable tasks are determined and ordered such that the task most likely to lead to the smallest makespan is placed first in the list. The partial schedule is then extended with the first element from this 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 already obtained. It goes back to the last decision point where there are still placeable tasks left and takes the next task in the ordered list. 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 shortest one obtained until then. In a cyclo-static graph, we call a task-instance placeable when all instances with lower index are scheduled and enough input data have been produced by predecessor tasks to fire the current instance as well. Formulas to check this mathematically are given in [6], while the heuristics used for ordering the tasks and calculating lower bounds can be found in [7].

## 3. CYCLO-STATIC DATA FLOW EXAMPLE

In this section we illustrate the importance of cyclo-static data flow for real-life DSP-applications with a video encoder. This encoder compresses video data for transmission through a wireless local area network and has been developed in collaboration with Prof. Meng's group at Stanford University. A prototype has been worked out on a C40-based multiprocessor [4]. Due to the very high sample-rate (0.8 million samples per second), dynamic scheduling could not be used. The cyclically changing behaviour of some tasks on the other hand made a description in synchronous data flow impractical. A way out to this problem consisted in the cyclo-static data flow approach.

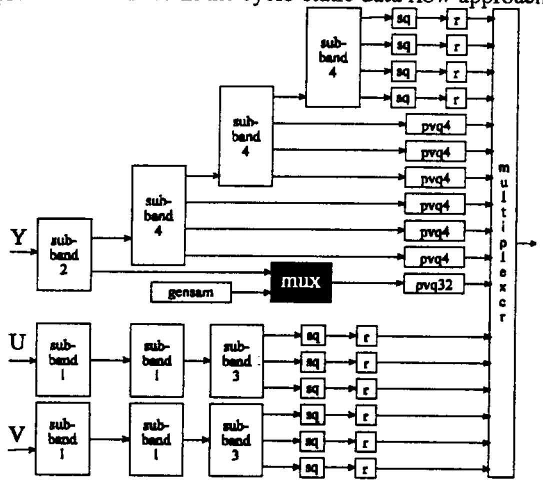

Fig. 4. Specification of the video encoder algorithm.

The basic structure of the encoding algorithm is given in fig. 4. The inputs to the system are the luminance (Y) and two chrominance components (U,V) of the video stream. The chrominance components were subsampled by a factor 2 in both spatial dimensions before being send to the encoder. (Not on the figure.) On each component, a subband decomposition is performed. The luminance component is decomposed into 13 bands of which 2 are zeroed out. The decomposition of each chrominance component contains 10 bands of which 7 are thrown away. For the low-frequency bands, the decomposition is followed by a scalar quantisation (sq). The higher-bands are pyramid vector quantised, respectively with vectors of 4 (pvq4) and 32 (pvq32) input samples. For the latter bands the length of the rows (originally 80 samples) are first made a multiple of 32 by adding 16 zero elements (gensam/mux) at the end of each row. To increase the error resiliency, the most-significant bits of the low frequency bands are repeated (r). Finally, all bands are combined in one bit-stream that is send to the radio interface.

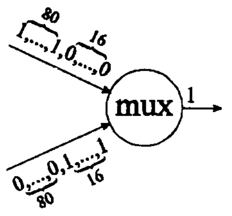

Fig.5. Cyclo-static representation of mux.

Different cyclo-static tasks can be found in this algorithm. The two dimensional subsampling, for instance, decimates the datarate by a factor 4, using an alternation of horizontal and vertical filters. This filter alternation corresponds to a behaviour that is cyclo-static over two lines. Also the mux is cyclo-static: the first 80 invocations it outputs the subband data, the next 16 invocations it transfers the zero elements (fig. 5). Finally the multiplexer is cyclo-static with a very complicated repetition pattern whose length corresponds to 64 input lines.

## CONCLUSIONS

In this paper we presented the cyclo-static data flow paradigm. Cyclo-static data flow allows to construct an efficient static schedule for applications with a cyclically changing behaviour. We gave both necessary and sufficient conditions for such a schedule to exist and presented a scheduling method as it is implemented in GRAPE-II. Typical for this method is that it does not require the graph to be transformed to a single-rate equivalent. Finally we also gave a real-life example to show the importance of a cyclo-static data-flow specification for real-time prototyping.

## REFERENCES

[1] R. Lauwereins, M. Engels and J. A. Peperstraete, "GRAPEII: A tool for rapid prototyping of multi-rate asynchronous DSP applications on heterogeneous multiprocessors", Proc. of the 3rd. Int. Workshop on Rapid System Prototyping, Research Triangle Park, North Carolina, USA, pp. 24-17, June 23-25, 1992.

[2] J. Pino, S. Ha, E. Lee and J. Buck, "Software Synthesis for DSP Using Ptolemy", Journal on VLSI Signal Processing, special issue on Synthesis for DSP, 1993.

[3] E. A. Lee and 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.

[4] M. Engels and T. Meng, "Rapid Prototyping of a Real-Time Video Encoder", Proc. of the 5th Int. Workshop on Rapid System Prototyping, Grenoble, France, pp. 8-15, June 21-23, 1994.

[5] R.M. Karp, R.E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination and Queueing", SIAM Journal on Applied Mathematics, Vol. 14, No. 6, pp. 1390-1411, Nov. 1966.

[6] G. Bilsen, M. Engels, R. Lauwereins, J.A. Peperstraete, "Static Scheduling of Multi-Rate and Cyclo-Static DSP-Applications", VLSI Signal Processing, VII, IEEE Press, New York, pp. 137-146, 1994.

[7] 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.

## ACKNOWLEDGEMENTS

Greet Bilsen is a Research Assistant, Rudy Lauwereins a Senior Research Associate and Marc Engels a Senior Research Assistant of the Belgian National Fund for Scientific Research. Part of this work has been performed when he was at Stanford University in the wireless video group of Prof. Teresa Meng. His visit was sponsored by the Fulbright-Heys program, NATO and NFWO. The work presented in this paper is partially sponsored by the Belgian Interuniversity Pole of Attraction IUAP-50 and the ESPRIT project 6800 Retides. K.U.Leuven-ESAT is a member of the DSP-Valley ™ network.