SCHEDULING FOR OPTIMUM DATA MEMORY COMPACTION IN BLOCK DIAGRAM ORIENTED SOFTWARE SYNTHESIS

Sebastian Ritz *Markus Willems* *Heinrich Meyr*
611810 ISS, RWTH, 52056 Aachen, Germany

ABSTRACT

For the design of complex digital signal processing systems, block diagram oriented synthesis of real time software for programmable target processors has become an important design aid. The synthesis approach discussed in this paper is based on multirate block diagrams with scalable synchronous dataflow (SSDF) semantics. For this class of dataflow graphs we present scheduling techniques for optimum data memory compaction. These techniques can be employed to map signals of a block diagram onto a minimum data memory space. In order to formalize the data memory compaction problem, we first derive appropriate implementation measures. Based on these implementation measures it can be shown that optimum data memory compaction consists of optimum scheduling as well as optimum memory allocation. For the class of single appearance (SA) block diagrams with SSDF semantics, scheduling can be reduced to an integer linear programming (ILP) problem. Due to the computational complexity of ILP, we also present a suboptimum scheduling selection criterion, which can be used for SA and non SA-schedulers.

1. INTRODUCTION

Memory compaction is an important optimization technique for systems with memory resource constraints. Especially in the implementation of digital signal processing systems we often find memory constraints due to limited available on-chip memory of programmable target architectures.

In this paper we focus on data memory compaction in the context of the synthesis of real time software using a block diagram specification of a signal processing system. As target processors digital signal processors (DSPs) are of special interest because of architectural features tailored to the specific needs of signal processing tasks. DSPs have stringent on-chip program memory limits and off-chip memory access is in general inefficient. In case of commercially available DSP cores, the on-chip RAM/ROM memory space may be adapted to the application specific needs. Since the size of the required memory space determines the costs, it is important to minimize the required memory space.

The block diagrams used for software synthesis are dataflow oriented and consist of blocks and signals. From the implementational point of view blocks are software modules (supplied by the user or the system) and signals are FIFO buffers in the data memory space. In case each block of a block diagram consumes and produces a fixed number of data samples, the block diagram is based on the "synchronous data flow" (SDF) [1] paradigm. These numbers, called *rates* in the sequel, must be specified a priori, e.g. at configuration time. Up- or downsampling within a block results in multi rate block diagrams. If all blocks of a block diagram may consume and produce any integer multiple of the predefined SDF-rates *per activation*, we call the SDF graph *scalable*, resulting in a scalable synchronous dataflow (SSDF) graph [2]. Due to the scalability, SSDF block diagrams can be optimally *vectorized* [3]. Vectorization in the context of SSDF graphs is regarded as a transformation on an SSDF graph raising the number of consumed and produced samples per activation to a certain integer multiple of the predefined SDF-rates. Because of the instruction and/or arithmetic pipelining of DSPs, vectorization leads to enhanced throughput of the synthesized software. Because of the increased vector lengths, vectorization also increases data memory consumption, which can be drastically reduced by the proposed data memory compaction.

The heuristic minimization of data memory consumption by means of looped schedules for SDF block diagrams has been discussed in [4]. The approach therein minimizes the vectorization opportunities for each block, thus is applicable for applications where throughput is not the primary optimization goal.

In section 2 we will introduce some of the basic formalisms of SSDF graphs. It follows the presentation of some implementation measures which serve as optimization criterions. In section 4 the optimum scheduling problem is treated. Afterwards we derive a suboptimum scheduling heuristics and finally demonstrate the scheduling strategies by means of an application example.

2. SCALABLE SYNCHRONOUS DATAFLOW: BACKGROUND AND NOTATION

We suppose that a digital signal processing system is specified by means of a scalable synchronous block diagram $F = (B, S, A, E, D_0)$. A block $b_j \in B$ specifies a signal processing component of arbitrary granularity. The signals $s_i \in S$ specify the data flow between the blocks. The topology of the block diagram is represented by the functions $A()$ and $E()$ defined on the signal set S. $A(s_i)$ is the block producing samples which are written in s_i and $E(s_i)$ is the block consuming samples which are stored in s_i and have been produced by $A(s_i)$. The number of initial samples on a signal s_i are specified by $D_0(s_i)$. An initial sample represents a phase shift of one sample. According to the synchronous data flow semantics a block $b_j = A(s_i)$ produces $O(s_i)$ samples written into signal s_i and consumes $I(s_k)$ samples from the signals s_k, where $b_j = E(s_k)$. Additionally in the SSDF domain a block may consume and produce any integer multiple $n(b_j)$ of the predefined rates, where $n(b_j)$ denotes the local blocking factor associated with block b_j.

A *schedule* Φ of an SSDF block diagram F is an activation sequence $\Phi = \{a_1, a_2, \ldots, a_p\}$ of blocks, where an activation $a_k = (b_j, n(b_j))$ either denotes the activation of a block with the local blocking factor $n(b_j)$ or the activation of a subschedule Φ_l with a looping factor k_l, $a_k = (\Phi_l, k_l)$. The latter describes a hierarchical schedule, which may be implemented with nested loops [2,4]. The looping factor k_l denotes the number of repetitions of the subschedule Φ_l. In the following we assume that all activations of a schedule Φ are *valid* in the sense that at the time of the activation of a block b_j there are at least as many samples at each input port as the block requires according to rates $I(s_i), \forall s_i : E(s_i) = b_j$.

0-7803-2431-5/95 $4.00

The activation $a_k = (b_j, n(b_j))$ of a block b_j is called an *appearance* of block b_j. If every block b_k appears exactly once in Φ, the schedule is called **single appearance** schedule (SAS). In case of a SAS each block has a unique local blocking factor $n(b_j)$.

In this paper we are interested in infinite schedules, where each block is activated infinitely often. For the software synthesis, we have to guarantee that an infinite valid schedule can be implemented with finite memory space for each signal. A sufficient condition for finite memory is that each block b_j of a scalable synchronous block diagram F is executed at least $q_F(b_j)$ times for one schedule period Φ [1], which can be repeated infinitely often. This condition ensures that in each schedule period as many samples are written to as are read from each signal. For multirate block diagrams for at least one block $q_F(b_j) > 1$ holds.

The *vectorization* of a given scalable synchronous block diagram determines the local blocking factor $n(b_j)$ for each block b_j. Since there is a unique local blocking factor for each block in case of SA-schedules, vectorization can also be regarded as a transformation on F increasing the rates of the input and output ports of the blocks. The vectorization is *valid*, if after vectorization there still exists a valid schedule. For all blocks the minimum number of executions $q_F(b_j)$ per schedule period can be increased by the *global* blocking factor N_g, which describes the global vectorization degree. In the sequel we assume that the block diagram has an associated SA-schedule and is vectorized, i.e. each block has a local blocking factor assigned. Also we restrict the discussion on flat SA-schedules, which means that we do not consider looped schedules.

3. IMPLEMENTATION MEASURES FOR DATA MEMORY CONSUMPTION

In order to derive optimum strategies for data memory compaction we first have to define the optimization criterion. Let $d_i(k)$ denote the number of signal samples in signal s_i in scheduling step k, $1 \leq k \leq p$. Then

$$M_{max}(\Phi) = \sum_{1 \leq i \leq N_s} \max_k \{d_i(k)\} \tag{1}$$

describes the total memory needed in case we allocate memory for each signal separately. Each signal s_i needs at least $\max_k \{d_i(k)\}$ memory space. For single appearance schedules and for signals s_i with $D_0(s_i) = 0$

$$\max_k \{d_i(k)\} = q_F(b_j) O(s_i) N_g \qquad b_j = A(s_i) \tag{2}$$

holds, since there is one write access followed by one read access on signal s_i per schedule period. Thereby p denotes the length of the period, which is equal to the number of blocks in case of SA-schedules and N_s denotes the total number of signals. For signals with $D_0(s_i) > 0$, the highwater $\max_k \{d_i(k)\}$ depends on whether due to the schedule Φ the first access to the buffer is read or write:

$$\max_k \{d_i(k)\} = \begin{cases} D_0(s_i) + q_F(b_j) O(s_i) N_g & \text{if first write} \\ D_0(s_i) & \text{if first read} \end{cases} \tag{3}$$

Note that the condition $D_0(s_i) \geq q_F(b_j) O(s_i) N_g$ has to be fulfilled in case the first access is a read, i.e. $E(s_i)$ is activated before block $A(s_i)$.

Signals with $D_0(s_i) = 0$ can be mapped onto *static* buffers of length $M(s_i) = q_F(b_j) O(s_i) N_g$. A buffer is said to be *static* iff the write resp. read access of the incident block occurs at the same memory offset in each schedule period. Static buffers can be efficiently synthesized, since blocks accessing those buffers just need a constant pointer to the memory segment allocated for the buffers. On the other hand, signals with $D_0(s_i) > 0$ i.g. can mapped only onto *dynamic* circular buffers of length $M(s_i) = D_0(s_i) + q_F(b_j) O(s_i) N_g$ or $M(s_i) = D_0(s_i)$, i.e. read and write accesses occur at different offsets from one period to the next. Dynamic buffers i.g. require offset computation at runtime thus exhibiting runtime overhead.

For both cases, $D_0(s_i) > 0$ and $D_0(s_i) = 0$, the implementation measure $M_{max}(\Phi)$ describes the expected data memory consumption for a given schedule Φ in case buffers use mutually exclusive memory spaces. From eq.(3) and (2) it can be seen that there is an optimization potential in evaluating a schedule Φ such that the total amount of memory needed for the signals s_i with $D_0(s_i) > 0$ is minimized by activating the proper reading blocks first. It can be shown that this optimization problem is a nonlinear scheduling problem. Note that the memory consumption can be optimized only by minimizing the lengths of the buffers corresponding to these signals and not by sharing memory between dynamic buffers. Dynamic buffers i.g. can not be mapped onto shared memory segments since write and read accesses are scattered to the whole buffer. In the sequel we will concentrate on optimally sharing memory for signals with $D_0(s_i) = 0$. Signals s_i for which $D_0(s_i) = 0$ holds will be denoted as s_i'. The optimization problem is regarded as more important, since i.g. there are much more signals without initial samples, especially after transformations like retiming, where explicitly initial samples are concentrated such that vectorization is optimized [5].

In order to take the effects of shared buffers into account another implementation measure is of interest. The number of signal samples present in all signals s_i with $D_0(s_i) = 0$ at schedule step k after activation $a_k = (b_j, n(b_j))$ can be described with the number of live signal samples $M_l(k)$, $1 \leq k \leq p$:

$$M_l(k) = \sum_{\forall s_i'} d_j(k) = M_l(k-1) + M_{out}(b_j) - M_{in}(b_j) \tag{4}$$

where $M_l(0) = 0$.

$M_{out}(b_j)$ denotes the total number of output samples produced on all signals s_i' upon activation of block $b_j = A(s_i')$:

$$M_{out}(b_j) = n(b_j) \sum_{\forall A(s_i') = b_j} O(s_i) \tag{5}$$

and $M_{in}(b_j)$ denotes the total number of input samples consumed from all signals s_i' upon activation of block $b_j = E(s_i')$:

$$M_{in}(b_j) = n(b_j) \sum_{\forall E(s_i') = b_j} I(s_i) \tag{6}$$

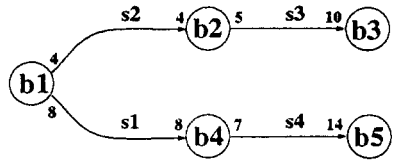
Note that $n(b_j) = q_F(b_j)$, since Φ is a flat SA-schedule. In case of non-inplace computation all input and output signal buffers have to be mapped onto disjunct memory spaces. Thus upon activation of block b_j we need $M_{out}(b_j)$ additional memory space for the storage of output samples. Note that signals s_i with $D_0(s_i) > 0$ are assumed to have separately allocated memory. It follows that for each activation $a_k \in \Phi$ there must be at least

$$M_{act}(a_k) = M_l(k-1) + M_{out}(b_j), \qquad a_k = (b_j, n(b_j)) \tag{7}$$

memory space. Taking the maximum of all $M_{act}(a_k)$ during a schedule period, leads to the implementation measure $M_{act}(\Phi)$:

$$M_{act}(\Phi) = \max_{a_k \in \Phi} \{M_{act}(a_k)\} \tag{8}$$

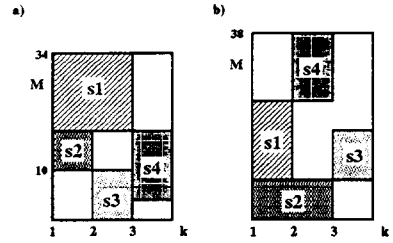
This implementation measure describes the minimum achievable amount of data memory for the signals s_i with $D_0(s_i) = 0$ in case all blocks do not process signal samples in-place. This implementation measure thus may serve as an optimization criterion for finding an optimum schedule, i.e. a schedule Φ which minimizes $M_{act}(\Phi)$. Fig. 1 shows a block diagram with two associated SA-schedules.



a)$\Phi_1 = \{2b_1, 2b_2, b_3, 2b_4, b_5\} \Rightarrow M_{act}(\Phi_1) = 34$
b)$\Phi_2 = \{2b_1, 2b_4, b_5, 2b_2, b_3\} \Rightarrow M_{act}(\Phi_2) = 38$

Figure 1. Influence of the scheduling on M_{act}

In a second step the signals of the block diagram have to be mapped onto memory segments. In fig. 2 the signals of the block diagram of fig. 1 are mapped onto memory according to the live times which can be determined by the schedule. For both schedules exactly $M_{act}(\Phi_{1/2})$ memory is needed.



a)$\Phi_a = \{2b_1, 2b_2, b_3, 2b_4, b_5\}$, non-inplace
b)$\Phi_b = \{2b_1, 2b_4, b_5, 2b_2, b_3\}$, non-inplace

Figure 2. Mapping of signals onto memory segments

Notice that the optimum data memory compaction is a two-step optimization problem. First a schedule has to be found which yields minimum $M_{act}(\Phi)$ and second all signal buffers have to be mapped onto memory such that buffers optimally share memory and $M_{act}(\Phi)$ is the amount of data memory needed. In the sequel we derive the optimum scheduling problem. Optimum memory allocation based on optimum scheduling will be presented in a forthcoming paper.

4. SCHEDULING FOR OPTIMUM DATA MEMORY COMPACTION

Given a block diagram F for which an optimum SA-schedule is to be found. In order to derive the number of live samples after schedule step n we define the cost matrix $\Gamma = (c_{ij})$ for all signals with $D_0(s_i) = 0$:

$$c_{ij} = \begin{cases} -q_F(b_j)I(s_i) & \text{if } E(s_i) = b_j \wedge D_0(s_i) = 0 \\ +q_F(b_j)O(s_i) & \text{if } A(s_i) = b_j \wedge D_0(s_i) = 0 \\ 0 & \text{else} \end{cases}$$

For the evaluation of the schedule we define the schedule variables $a_{j,k}$, $1 \le j,k \le N_{bl} = p$:

$$a_{j,k} = \begin{cases} 1 & \text{if block } b_j \text{ is activated at k} \\ 0 & \text{else} \end{cases}$$

The number of live samples $M_l(n)$ in all signals s_i with $D_0(s_i) = 0$ after schedule step n can be computed now by means of the cost matrix Γ and the schedule variables:

$$M_l(n) = \sum_{i=1}^{N_s} \sum_{j=1}^{N_{bl}} c_{ij} \left(\sum_{k=1}^{n} a_{j,k} \right) \qquad 1 \le n \le p \qquad (9)$$

For the evaluation of $M_{act}(\Phi)$ we further define the cost matrix $\Gamma' = (c'_{ij})$:

$$c'_{ij} = \begin{cases} O(s_i)q_F(b_j) & A(s_i) = b_j \wedge D_0(s_i) = 0 \\ 0 & \text{else} \end{cases}$$

With this cost matrix we can describe the implementation measure $M_{act}(\Phi)$:

$$M_{act}(\Phi) = \max_{1 \le n \le N_{bl}} \left\{ M_l(n-1) + \mathbf{I}^T \Gamma' \mathbf{a}(n) \right\} \qquad (10)$$

The vector $\mathbf{I}$ denotes the unity vector and $\mathbf{a}(n)$ the scheduling vector $\mathbf{a}(n) = (a_{1,n}, a_{2,n}, \ldots, a_{N_{bl},n})^T$. Goal of the optimization is to find an optimum schedule Φ_{opt} such that

$$M_{act}(\Phi_{opt}) = \min_{\text{all } \Phi} \{M_{act}(\Phi)\}$$

In order to linearize this min-max optimization criterion, we introduce a new variable $MACT$ together with N_{bl} constraints:

$$\text{minimize } MACT \qquad (11)$$

$$\text{such that for } 1 \le n \le N_{bl}$$

$$MACT \ge \sum_{i=1}^{N_s} \sum_{j=1}^{N_{bl}} c_{ij} \left(\sum_{k=1}^{n-1} a_{j,k} \right) + \sum_{i=1}^{N_s} \sum_{j=1}^{N_{bl}} c'_{ij} a_{j,n} \qquad (12)$$

holds. Beside this linear optimization criterion we have to derive linear constraints on the schedule variables. The first class of constraints can be derived from the fact, that Φ is a SASA-schedule:

$$\sum_{k=1}^{N_{bl}} a_{j,k} = 1 \qquad 1 \le j \le N_{bl} \qquad (13)$$

Since we are interested in a sequential schedule, only one block may be activated at time k:

$$\sum_{j=1}^{N_{bl}} a_{j,k} = 1, \qquad 1 \le k \le N_{bl} \qquad (14)$$

The last class of constraints follows from the signals of the block diagram, which can be regarded as simple precedence relations between adjacent blocks $b_l = A(s_i), b_j = E(s_i)$, since F has an associated SA-schedule:

$$\sum_{k=1}^{N_{bl}} (k a_{l,k} - k a_{j,k}) \le -1 \qquad \forall s_i : D_0(s_i) < n(b_j) q_F(b_j) I(s_i) \qquad (15)$$

Thus optimization criterion 4 together with the constraints 4, 13 and 14 form an integer linear programming problem, which can be solved using standard software packages.

Since an ILP is np-hard, we introduce a novel scheduling heuristics, which is derived from the above ILP.

5. SUBOPTIMUM SCHEDULING FOR MINIMUM DATA MEMORY CONSUMPTION

In the following we present a scheduling criterion which decides at time k which of the blocks to activate next. This criterion can be used for S-class scheduling algorithms which successively schedule a block depending on whether there are enough input samples available for a block. Although we restrict the discussion to SA-schedules, the presented criterion can also be used for multiple appearance schedules.

Due to eq. (4) and (7) scheduling a block b_j at step n determines the number of live samples $M_l(n)$ and the number of output samples for which additional memory space has to be allocated. Thus if at time $n-1$ more than one block can be activated, we have to select one of these blocks in a way that $M_{act}(\Phi)$ is minimized.

Given now a candidate set of blocks

$$C = \{b_j \mid \forall_{s_i} E(s_i) = b_j : d_i(n-1) \ge O(s_i)\} \qquad (16)$$

which all may be activated at time n, we can split this set into two sets. The first set of blocks includes all blocks whose activation does not increase the number of live samples, i.e. $M_l(k) \leq M_l(k-1)$, $k > n$:

$$C_1 = \{b_j \mid b_j \in C \wedge M_{in}(b_j) \geq M_{out}(b_j)\} \quad (17)$$

The second set C_2 includes all blocks which increase the number of live samples upon their activation, i.e. $M_l(k) > M_l(k-1)$, $k > n$:

$$C_2 = \{b_j \mid b_j \in C \wedge M_{in}(b_j) < M_{out}(b_j)\} \quad (18)$$

We now regard all $\|C\|!$ permutations of blocks of C which form a valid possible subschedule

$$\Phi_m = \{a_k \mid a_k = (b_j, n(b_j)), b_j \in C\}$$

for blocks of C and ignore the previous activations a_k with $k \leq n$. Then the subschedule $\Phi_{opt} = \{\Phi_1, \Phi_2\}$ with

$$\Phi_1 = \{a_k \mid a_k = (b_j, n(b_j)), b_j \in C_1\} \quad (19)$$

$$\Phi_2 = \{a_k \mid a_k = (b_j, n(b_j)), b_j \in C_2\} \quad (20)$$

exhibits the minimum number of maximum live samples:

$$\max_{a_k \in \Phi_{opt}} \{M_l(k)\} = M_l(n) + \sum_{b_j \in C} M_{out}(b_j) - \sum_{b_j \in C} M_{in}(b_j)$$

$$\leq \max_{\Phi_m \neq \Phi_{opt}} \{M_l(k)\} \quad n \leq k \leq n + \|C\|$$

Since we have only regarded all subschedules Φ_m this might be a local but not a global minimum.

Now the question arises how to minimize $M_{act}(\Phi_m)$ of all subschedules Φ_m. This minimization can be achieved by sorting the blocks $b_j, b_i \in C_1$ according to $M_{out}(b_j)$:

$$M_{out}(b_j) \geq M_{out}(b_i) \Rightarrow b_i \text{ before } b_j \quad (21)$$

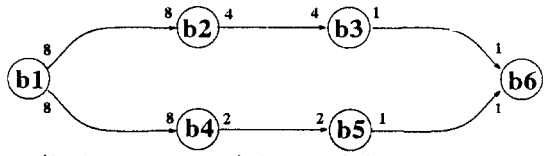
It can be shown that this sorting is sufficient for obtaining minimum $M_{act}(\Phi_{1m})$ concerning all permutations $\|C_1\|!$ of valid schedules Φ_{1m} of blocks in C_1.

The same can be shown for the second set C_2. Sorting all blocks of C_2 with

$$M_{in}(b_j) \geq M_{in}(b_i) \Rightarrow b_j \text{ before } b_i \quad (22)$$

yields minimum $M_{act}(\Phi_{2m})$ concerning all permutations $\|C_2\|!$ of valid schedules Φ_{2m}.

Thus a scheduling algorithm based on this heuristic minimization criterion simply has to insert each successor of the last scheduled block which can be activated into C_1 or C_2 and has to sort within these sets according to rule 21 or rule 22. In figure 3 an example for such a heuristic schedule algorithm is shown. At $n = 1$, $C_1 = \{b_2, b_4\}$ and C_2 is empty. Since $M_{out}(b_4) = 2 \leq M_{out}(b_2) = 4$, block b_4 is scheduled before b_2. In the next step $n = 2$, $C_1 = \{b_2, b_5\}$ holds. Since $M_{out}(b_5) = 1 \leq M_{out}(b_2) = 4$, b_5 is scheduled next. This schedule is also the optimum one.



$\Phi_{opt} = \{b_1, b_4, b_5, b_2, b_3, b_6\} \Rightarrow M_{act}(\Phi) = 18$

Figure 3. Optimum schedule through the heuristic schedule criterion

An example for the suboptimality of the selection criterion can be seen in the schedule for the block diagram of figure 1. At $n = 1$, $C_1 = \{b_4\}$ and $C_2 = \{b_2\}$. Following the above selection criterion, block b_4 is scheduled before b_2, which is suboptimal.

6. APPLICATION

As an example for the derived scheduling methods, a realistic application example is given. In fig.4, the block diagram of a mobile satellite receiver is shown [6] .

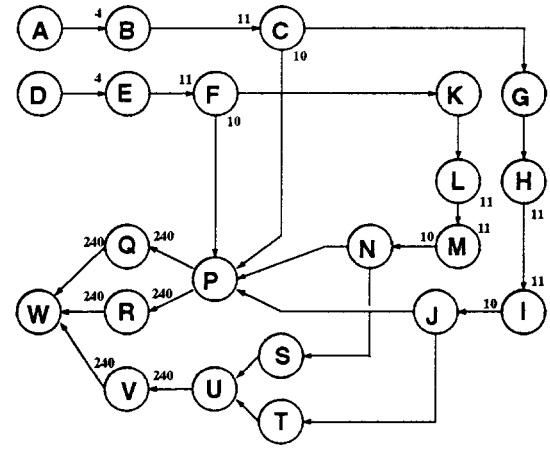


Figure 4. Rates within the mobile satellite receiver block diagram

The heuristic scheduling results in
$\Phi_1 = 1056A, 264B, 24C, 24G, 24H, 24I, 240J, 1056D, 264E, 24F, 24K, 24L, 24M, 240N, 240P, 240S, 240U, V, Q, R, W$,
requiring 2040 words of memory.
One optimum scheduling sequence (there are several) is
$\Phi_2 = 1056A, 264B, 24C, 24G, 1056D, 264E, 24F, 24K, 24L, 24M, 240N, 24H, 24I, 240J, 240P, 240S, 240U, V, Q, R, W$,
requiring 1920 words of memory. It took about 4 days to solve the ILP problem, whereas the heuristic approach delivered its result within 30 seconds.

7. CONCLUSION

We have presented a novel optimum scheduling approach resulting in a minimum of data memory consumption for single appearance constellations. This approach has been identified as an ILP problem, driving us to define an efficient heuristic. For a realistic application this heuristic has been shown to result in a minor degradation of data memory efficiency, offering a reliable complexity estimation within a significantly reduced amount of time.

REFERENCES

[1] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. on Computers*, vol. C-36, No. 1, pp. 24–35, January 1987.

[2] S. Ritz, M. Pankert, and H. Meyr, "High level software synthesis for signal processing systems," in *Proceedings of the Intl. Conf. on Application-Specific Array Processors*, pp. 679–693, Prentice Hall, IEEE Computer Society, 1992.

[3] S. Ritz, M. Pankert, V. Živojnović, and H. Meyr, "Optimum vectorization of scalable synchronous dataflow graphs," in *Intl. Conf. on Application-Specific Array Processors*, pp. 285–296, Prentice Hall, IEEE Computer Society, 1993.

[4] S. Bhattacharyya and E. Lee, "Scheduling synchronous dataflow graphs for efficient looping," *Journal of VLSI Signal Processing*, pp. 271 - 288, Dec. 1993.

[5] V. Živojnović, S. Ritz, and H. Meyr, "Retiming of DSP programs for optimum vectorization," in *Proc. of ICASSP'94 - Adelaide*, April 1994.

[6] S. Ritz and H. Meyr, "Exploring the design space of a DSP-based mobile satellite receiver," in *Proc. of ICSPAT'94 - Dallas*, Oct. 1994.