Cathedral-III : Architecture-Driven High-level Synthesis for High Throughput DSP Applications. *

Stefaan Note[1] Werner Geurts[1] Francky Catthoor[1,2] Hugo De Man[1,2]

[1]IMEC Laboratory, Kapeldreef 75, B-3001 Leuven, Belgium
[2]ESAT Laboratory, Katholieke Universiteit, K. Mercierlaan 94, B-3001 Leuven, Belgium

Abstract
The goal of this paper is to extend the synthesis of real time digital signal processing (DSP) algorithms towards the domain of high throughput applications. A novel architectural style specifically suited for this application domain is presented. Furthermore, a synopsis of a novel synthesis script typically oriented towards this architecture is described (architecture-driven synthesis). The emphasis in the script is on the design of the data-paths which are dedicated to the application, and special attention is paid to the memory synthesis problem. In this paper only the data-path related tasks, namely data-path partitioning and data-path definition, are discussed. The new methodology is demonstrated using an image processing application.

1 Introduction

This paper is focusing on the *synthesis of high throughput DSP applications* which can be found in the telecommunication, medium-end image and video processing domain. These applications are characterized by signal flow graphs which exhibit a *large amount of repetitivity* (due to *FOR, WHILE*-loops), *recursion* (occurring whenever a new time-loop iteration depends on the result of the previous iteration), *multi-dimensional signals*, and *non-linear operations* (such as absolute value and maximum/minimum calculations). Furthermore, the total number of operations to be performed per second is large ($> 100 Moperations/second$).

A first class of high-level synthesis systems is mainly tuned towards a *highly multiplexed architectural style* [1, 2], which is based on predefined highly programmable data-paths (such as ALUs). The target applications are *low sample rate, complex decision making algorithms* containing a large number of operations. For these applications, *scheduling/assignment is one of the most important synthesis tasks*. Heuristics have been developed to cope with scheduling large signal flow graphs ($100 - 5 * 10^5$ operations) assuming a large ($100 - 10^5$) number of clock cycles available. A limitation of this architectural style and the corresponding synthesis techniques is that *recursive bottlenecks*, require too many cycles to execute. This limits the maximal achievable sample frequency.

*Research supported by the "ESPRIT 2260" program of the E.C.

On the other hand, for high throughput applications, *pipeline scheduling techniques* have been proposed in [4, 5, 6]. Typically, the target architecture consist of a *large (5 .. 20) number of predefined functional units* (FUs). A FU is defined as an unit that can only perform a few (< 5) number of operation types. These approaches solve the high performance requirements by pipelining the time loop. As a result of pipelined scheduling, the DSP algorithm is partitioned into sequences of operations in such a way that they can be executed concurrently. Because the existing approaches, *do not take the repetitivity in the algorithm into account, the interconnection and register cost between the different FUs is large*. Furthermore, these approaches cannot handle in an efficient way recursive algorithms. The reason is that the architectural style and the software algorithms only allow limited chaining of operations.
Another interesting approach to the synthesis problem of high throughput DSP applications is the *HYPER* system [7]. The designer can define flexible data-paths onto which all the operations of the algorithm are scheduled. Emphasis in this environment is on transformations and scheduling.

The basis of this paper is the *lowly multiplexed cooperating data-path architectural style* [8], which is fully oriented towards the domain of high performance applications. The *fundamental principle* of this style in terms of data-paths is to base their *composition* on the *repetitivity*, and the *signal flow dependencies* of the signal flow graph representing the DSP algorithm. These dedicated data-paths are necessary in order to cope with the high throughput requirements of the applications, taking recursion and the non-linear operations of the algorithm into account. Since the ratio of the maximal achievable clock frequency and the sample frequency is low, ranging between 1 and 10, the allocated hardware resources are shared only for a limited number of clock cycles. This explains the term "lowly multiplexed". Figure 1 gives an example of a *processor architecture* composed out of *distributed memories*, *dedicated data-paths* and a *controller*. It shows the typical internal structure of a dedicated data-path composed out of ABBs. An *abstract building block* (ABB) is an abstract type of primitive operator at the RT-level for which one or more hardware representations exist in a hardware library. An example is the ABB adder, which can be realized as either a ripple-carry, a carry-bypass or a lookahead adder. Note in figure 1 the different ABBs which are chained together. Due to the dedicated composition and interconnection of the ABBs, *critical parts* of the algorithm can be *evaluated in less cycles* than is possible on a predefined, highly programmable data-path such as a arithmetic logic unit. Furthermore, the critical path of the dedicated data-path is only slightly larger than the delay of a single ABB, since

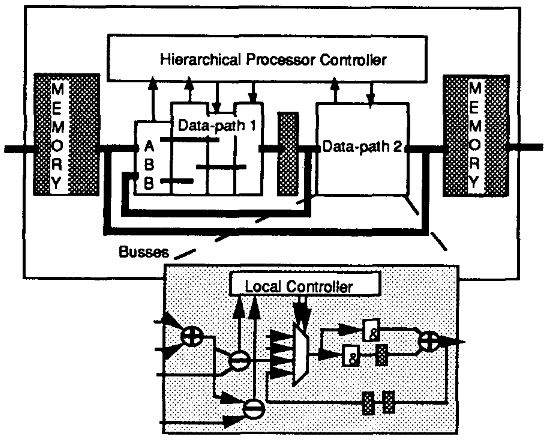


Figure 1: *Processor architecture and a dedicated data-path.*

	HSF	+	*	*mux*	*reg*
Park [5]	3	5	3	23	42
GE. [6]	2	8	4	20	18
C-III	2	8	4	3	4

Table 1: *Comparison of the results of traditional approaches and our C-III approach.*

the propagation delay has to be accounted only once [9]. Finally, because of the local connections inside the data-path, *global interconnections* between the different data-paths are *reduced significantly.* Such a customized data-path is referred to as an application specific unit (ASU).

Since in the lowly multiplexed data-path architecture [10], the internal composition of the data-paths is not predefined, they have to be defined "on the fly" during synthesis. As a consequence, one of the *main high-level synthesis requirements is the definition and optimization of these data-paths (ASUs)* starting from a high-level description of the complete algorithm. Furthermore, because of the high data rates and the multi-dimensional signals, *memory management* is an important design task.

Table 1 illustrates the considerable savings in terms of multiplexers and registers obtained with our approach compared to the results of [5, 6] for the 16 tap symmetrical FIR filter [4] [1]. Figure 2 shows our architecture based on dedicated data-paths. In order to arrive to this efficient solution, first the data-paths are defined and then the operations are scheduled on these data-paths. By optimizing the data-paths [9], the final clock frequency can be made the same as in [5, 6].

2 The Cathedral-III design script

The synthesis process is divided into 2 main tasks : behavioural synthesis, and structural synthesis. Figure 3 gives a global overview of the *CATHEDRAL-III* script.

2.1 Behavioural synthesis process

The input of this task is a description of the behaviour of the algorithm. For our purposes the applicative language *SILAGE* [11] is used. Behavioural synthesis produces an architecture netlist while iterating over the following steps.

[1] For a definition of *HSF* see section 3.1

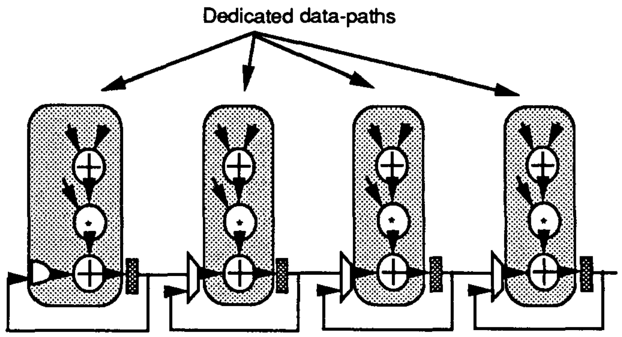


Figure 2: *Cathedral-III architecture for the 16 tap symmetrical filter.*

1. *High-level memory management :*
The *main tasks* are : (i) the determination of the *number and type of back-ground memory* units, (ii) the selection of the relative address locations in one of the allocated memories to store the signals. A back-ground memory is a large memory (RAM, FIFO) required for storing large multi-dimensional signals such as speech frames or parts of images. The *objectives* are three-fold : (i) providing *sufficient memories and memory ports* so that the communication with the memory does not limit the performance requirement, (ii) *reduce* the total *amount of memory,* and (iii) *reduce* the complexity of the *addressing units.* Since memories constitute more than 50% of the total size of chip, it is *important first to look at the memory organization.* As a result, additional constraints occur which have to be taken into account during the following data-path synthesis step.
2. *High-level data-path mapping tasks :*
This main task involves different subtasks related with the definition and generation of the customized data-paths.

Before the data-paths are generated, the *signal flow graph description is optimized.* The optimizations considered here affect the composition and the performance of the resulting data-paths. A specific DSP transformation is the combination of retiming and tree-height reduction.

Based on the ratio of the estimation of the clock frequency f_c and the given sample frequency f_s, the signal flow graph is *partitioned into clusters* (groups of 1..20 operations) which are grouped into one or more sets, and for each set a *dedicated data-path is defined* which can execute each of the clusters of the set in one cycle. The first task is referred to as ASU-partitioning (section 3.1), and the second as ASU-definition (section 3.2).

Each cluster in the signal flow graph is assigned to an ASU instance, and is substituted by 1 superoperation. The resulting graph is a *graph of superoperations.* After *scheduling* this graph, the total number of clock cycles *#cycles* is derived.

The next step involves *optimization of the data-paths* [9] given the clock frequency estimate. If the throughput requirement f_c cannot be satisfied, *iterations over the high-level data-path mapping tasks* are required : other clusters are defined which are grouped in an alternative way, the data-paths are pipelined (multi-cycle operations) which can affect the schedule.
3. *Low-level memory management :*
Low-level memory management involves the allocation of fore-ground memory and the selection of memory locations for *temporarily storing signals.* Examples are individual registers and register files.

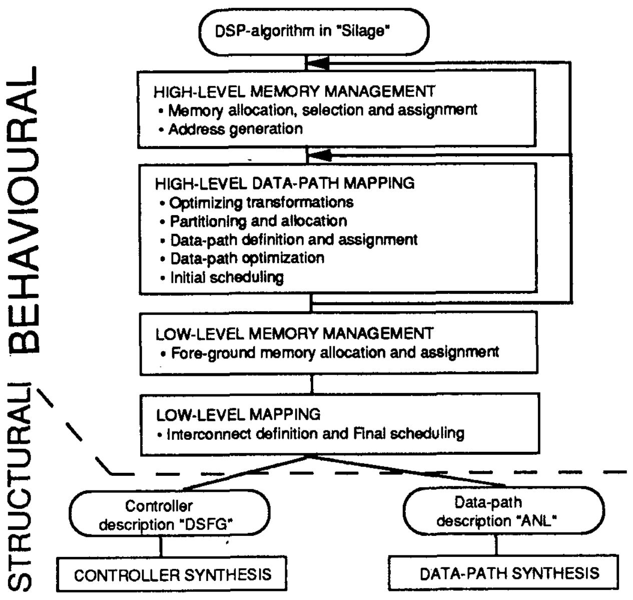

DSP-algorithm in "Silage"
HIGH-LEVEL MEMORY MANAGEMENT
• Memory allocation, selection and assignment
• Address generation
HIGH-LEVEL DATA-PATH MAPPING
• Optimizing transformations
• Partitioning and allocation
• Data-path definition and assignment
• Data-path optimization
• Initial scheduling
LOW-LEVEL MEMORY MANAGEMENT
• Fore-ground memory allocation and assignment
LOW-LEVEL MAPPING
• Interconnect definition and Final scheduling
Controller description "DSFG"
Data-path description "ANL"
CONTROLLER SYNTHESIS
DATA-PATH SYNTHESIS
BEHAVIOURAL
STRUCTURAL

Figure 3: *Global CATHEDRAL-III design script.*

4. *Low-level data-path mapping tasks :*
Low-level mapping mainly involves the detailed scheduling, and the detailed interconnect generation.

2.2 Structural synthesis process

Structural synthesis involves the detailed netlist generation of the data-paths and the controllers. These netlists are the input for layout generation programs such as data-path assemblers and general cell place and route systems.

3 ASU-partitioning and definition.

3.1 ASU-partitioning.

Before a particular strategy for partitioning a signal flow graph is described, the following 4 definitions are given.

Definition 1 *A cluster cl_i is a connected subgraph $g_i(v, e)$ of the complete signal flow graph $G(V, E)$ with vertices v corresponding to arithmetic/logic/relational operations and edges e corresponding to signals.*

Definition 2 *A set $S = \{cl_1, ..., cl_k\}$ is a collection of k clusters for which a dedicated data-path has to be constructed that can execute the clusters each in only one cycle.*

Each cluster has a multiplicity m_i corresponding to the number of times a cluster will be executed during the execution of the algorithm. The multiplicities m_i are calculated from the parameters of the loops surrounding the cluster cl_i (excluding the infinite time loop).
Assume the algorithm has a maximal hardware sharing factor HSF_{max} defined as the *ratio* of the interval available to execute the algorithm for one set of input data periodically, $T_s = 1/f_s$, and the clock period $T_c = 1/f_c$. This means that the hardware resources allocated for the algorithm can only be shared HSF_{max} times.

Definition 3 *The size of a set $S = \{cl_1, ..., cl_k\}$ with multiplicities $\{m_1, ..., m_k\}$ is defined as $\sum_{i=1}^{k} m_i = HSF_{act}$.*

Definition 4 *The cardinality of a set $S = \{cl_1, ..., cl_k\}$ is defined as the number of clusters in the set.*

In the context of the generation of application specific data-paths, the *goal of partitioning* is twofold :
(i) divide the complete signal flow graph $G(V, E)$ describing the algorithm into clusters $g_i(v, e)$ such that $\forall \nu \in V$: $\exists! g_i(v, e) : \nu \in v$.
(ii) determine the number of sets S_i and the clusters that belong to the same set S_i such that the total area of the ASUs is minimized.
The *result of the partitioning* phase consists of different sets S_i determining the exact number of data-path instances allocated in the architecture netlist. *Each set corresponds to the specification of a dedicated data-path.* For most of the applications targeted in this work, the cardinality of a set S_i is relatively small < 10. Furthermore, the number of operations in a cluster is typically smaller than 20.

Partitioning the signal flow graph as a preprocessing step for ASU definition consists of the following 2 steps.

3.1.1 Defining clusters in a graph

The definition of clusters in a graph is mainly guided by the available *hierarchy and repetitivity* present in the high throughput DSP algorithm [10].

- The full body or part of the body of a *repetitive definition* such as a $FOR - loop$ or a $WHILE - loop$. Since such a body is executed a large number of times, it is worthwhile to evaluate this body as fast as required.
- The full body or part of the body of a function definition (hierarchy).

The result is a DSP algorithm which is partitioned into several clusters such that each operation in the DSP algorithm is contained in 1 and only 1 cluster.

3.1.2 Defining sets

In a second step, the *goal is to group the clusters into one or more sets* based on the compatibility of the clusters. Suppose C_a is the area cost of a data-path able to execute cluster cl_a, and C_{ab} is the area cost of a dedicated data-path able to execute both clusters cl_a and cl_b. The compatibility of 2 clusters is defined as the overhead cost of merging the 2 clusters on the same data-path : $C_M(a, b) = 2C_{ab} - C_a - C_b$ [13]. If $C_M = 0$, there is no area overhead as a result of merging 2 clusters on the same data-path. The costs C_a and C_b are respectively computed as the sum of the area of the ABBs required to implement the operations of clusters cl_a and cl_b. The cost C_{ab} is estimated using the algorithm proposed in section 3.2. A compatibility graph is constructed with vertices corresponding to all the clusters, and edges between each pair of vertices a and b with weight $C_M(a, b)$. This graph is partitioned into different sets in such a way that the total weight of the edges which do not cross the partitions is minimized, given the constraint that the size of each partition (set) must be smaller than HSF_{max}. The partitioning problem is formulated as a 0/1 quadratic programming formulation.

3.2 ASU-definition

The result of the partitioning process consists of one or more sets S_i each containing a number of clusters. *The goal of ASU-definition is to define for each set a dedicated data-path which can execute each cluster in the set in one*

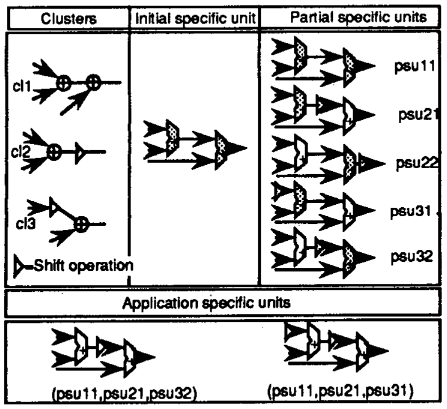


Figure 4: *Overview of the ASU generation process.*

cycle. The number of sets determines the total number of data-path instances. As a preprocessing step, high-level operations are expanded to primitive operations which are available in the operation-operator library. For example, the cluster $S^* = \{cl_1\} = \{(y = abs(a - b))\}$ becomes $S^* = \{cl_1\} = \{(tmp = a - b, cnd = a > b, out = cnd?tmp|0 - tmp)\}$. Furthermore, if a cluster contains conditional operations, it is expanded into several clusters. In the above example of the absolute value operation, 2 clusters are derived which are collected in the same set $S = \{cl_{1a}, cl_{1b}\}$: $cl_{1a} = (tmp = a - b; cnd = a >= b; out = tmp)$, and $cl_{1b} = (tmp = a - b; cnd = a >= b; out = 0 - tmp)$. The multiplicity of $cl_{1a} = cl_{1b} = cl_1/2$. In this section, we will describe how a lowly programmable, customized data-path is generated given a set S. A CAD tool *MOZART* has been developed which consists of the following 3 phases [10] :

1. Derive for the set S, the initial hardware requirements. This results in the specification of an *initial specific unit ISU*.
2. Assign all the clusters cl_i of the set S onto the defined initial specific unit ISU. Since different assignments of the operations $\in cl_i$ on the ISU are possible, this will result in a number of *partial specific units* PSU_{ij}.
3. Construct from all the partial specific units PSU_{ij} corresponding to the clusters cl_i, the final *application specific unit ASU*.

3.2.1 Initial Specific Unit Selection

The goal of this step is to define the initial composition of the data-path in terms of ABBs and their interconnection. The selection is derived from the most complex cluster for which a hard-wired data-path solution, ISU, is generated : a one-to-one mapping of operations to ABBs and signals to connections is performed. The complexity of a cluster is measured as the total area of the ABBs corresponding to the operations in the cluster. For example, figure 4 shows the initial specific unit derived from the most complex cluster cl_1.

3.2.2 Partial Specific Unit Derivation

The goal of this subtask is to generate all possible assignments of the operations of the cluster to the ABBs of the initial specific unit. This process is repeated for all clusters in the set $S = \{cl_1, cl_2, ..., cl_n\}$. This results in the following set of partial specific units $P = \{(psu_{11}, psu_{12}, ..., psu_{1k}), (psu_{21}, ..., psu_{2l}), ...\}$.
Each partial specific unit psu_{ij} is specified in the following way :

- The i index refers to the cluster cl_i for which different assignments on the ISU are generated. The j index refers to the different assignments which are possible.
- The assignment of every operation and signal in the cluster cl_i to respectively the ABBs and the terminals of the ABBs of the initial specific unit. From this, the corresponding operation modes of all the ABBs are derived.

Since it is unlikely that all the clusters can be assigned on the same hard-wired data-path additional programmability (constructive) is added in one of the following ways :

- *The functionality of the ABBs is dynamically extended.* Consider for example the ABB "adder" whose functionality is extended to become a programmable adder/subtractor.
- *One input/one output ABBs are added to the initial data-path.* Consider for example a programmable shifter which is added to the initial data-path.
- *Multiplexers are inserted at the inputs of the data-path.*

The basic idea of the technique is to make a data-path programmable by first trying to extend the functionality of certain ABBs, and adding additional ABBs, instead of solely adding multiplexers. This approach resembles the way human designers define a programmable data-path. The algorithm to generate these partial specific units is referred to as a the *constructive matching algorithm.* As shown in figure 4, the shaded ABBs in the partial specific unit psu_{ij} correspond to ABBs which have an operation of the cluster cl_i assigned to them. Note the shifters which have been added to the initial specific unit.

3.2.3 Application Specific Unit Generation

Only when for all clusters in the set S, all possible partial specific units are derived, the application specific unit can be generated. In a first step, all the possible combinations C of the partial specific units of all the clusters cl_i in the set S are enumerated. Assume we start from a set $S = \{cl_1, cl_2, cl_3\}$ which results in the following partial specific units :
$P = \{(psu_{11}), (psu_{21}, psu_{22}), (psu_{31}, psu_{32})\}$ (figure 4). The list of all possible combinations is $C = \{(psu_{11}, psu_{21}, psu_{31})\ (psu_{11}, psu_{21}, psu_{32}), (psu_{11}, psu_{22}, psu_{31}), (psu_{11}, psu_{22}, psu_{32})\}$.
Due to the constructive nature of the matching algorithm, all the partial specific units in one combination $\in C$ do not necessarily contain the same hardware resources. Therefore, for each combination $cb_k \in C$, all the ABBs which have been additionally allocated during the previous phase, are added to the initial unit. This results in a potential application specific unit ASU_k corresponding to the combination cb_k. For our example, figure 4 shows the shifters which have been allocated for both combinations. Finally, the area of all the ASU_k is estimated based on all the modes of the ABBs $\in ASU_k$. The ASU_k with the smallest area is selected. The complexity of this last step is NP-complete, however heuristics are defined which prune the number of combinations.

```
func MaxMin (in [][], max) maxmin[][] =
(i : 1 .. 512) ::
begin
 (j : 1 .. 512) ::
 begin
  tmp[i][j][-5] = 0;
  (m : -1 .. 1) ::
  begin
    (n : -1 .. 1) ::
    begin
     tmp[i][j][3m+n] =
      if (max)
       -> max (tmp[i][j][3m+n-1], in[i-m][j-n])
       || min (tmp[i][j][3m+n-1], in[i-m][j-n])
      fi;
    end;
   end;
   maxmin[i][j] = tmp[i][j][4];
 end;
end;
```

Figure 5: *SILAGE description of the max/min unit.*

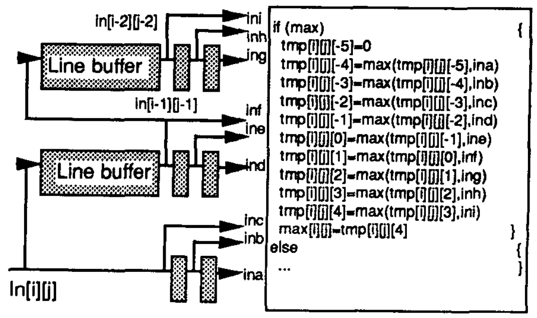
4 The ASIC synthesis process : an example.

The example is a 3*3 maximum/minimum unit used in an image processing environment [12]. The maximal required processing time for 1 image is 20 ms, which results in a sample frequency of $50Hz$. The behavioural description is expressed in the applicative language *SILAGE* (see figure 5).

The total number of operations to be performed per second is $512^2 * 3^2 * 50 = 117M$. If a clock frequency of $15MHz$ is assumed, the $HSF = 15MHz/50Hz = 3 * 10^5$. Given the $3*10^5$ clock cycles available, for the body of the most inner *n-loop* only 0.13 clock cycle is available. Because the number of clock cycles is smaller than 1, the inner loops m and n are unrolled (see figure 6). After this unrolling, 1 cycle is available to execute the body of the inner *j-loop*.

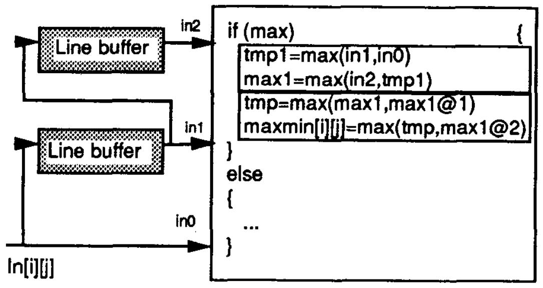
4.1 Behavioural synthesis process

1. *High-level memory management :*
   Since only 1 cycle is available to evaluate the body of the *j-loop*, all 9 pixels must be present at the same time. During the back-ground memory allocation phase, 2 line buffers are allocated. Figure 6 shows the overall memory organization, and the resulting pseudo *SILAGE* description for the body of the inner *j-loop*.
2. *High-level data-path mapping tasks :*
   1) Transformations of the signal flow graph.
   After transforming the body, instead of 9 max/min functions, only 4 are required. As a direct consequence also the critical path reduces from 9 comparisons to only 4 comparisons. Figure 7 shows the description of the body after the transformations. The delay operator @ is referring to a value computed at the previous time index. For example in figure 6, $inh = ini@1$ and $ing = inh@1 = ini@2$.
   2) Partitioning.
   During partitioning, the optimized body of the inner j loop is divided into 4 clusters. These 4 clusters contain only arithmetic and logic operations. Each cluster either corresponds to the maximum or minimum calculation of 3 values. Since 2 of the 4 clusters are mutually exclusive with the 2 other clusters, and since only 1 cycle is available, the 4 clusters are grouped into 2 sets S_1 and S_2 (see figure 8). For each set, a dedicated data-path (ASU) will be generated that depending on the value of the input max either calculates the maximum or minimum of 3 values.



```
if (max)                                   {
  tmp[i][j][-5]=0
  tmp[i][j][-4]=max(tmp[i][j][-5],ina)
  tmp[i][j][-3]=max(tmp[i][j][-4],inb)
  tmp[i][j][-2]=max(tmp[i][j][-3],inc)
  tmp[i][j][-1]=max(tmp[i][j][-2],ind)
  tmp[i][j][0]=max(tmp[i][j][-1],ine)
  tmp[i][j][1]=max(tmp[i][j][0],inf)
  tmp[i][j][2]=max(tmp[i][j][1],ing)
  tmp[i][j][3]=max(tmp[i][j][2],inh)
  tmp[i][j][4]=max(tmp[i][j][3],ini)
  max[i][j]=tmp[i][j][4]                   }
else                                       {
  ...                                      }
```

Figure 6: *Memory organization of the max/min unit.*



```
if (max)                          {
  tmp1=max(in1,in0)
  max1=max(in2,tmp1)
  tmp=max(max1,max1@1)
  maxmin[i][j]=max(tmp,max1@2)
}
else
{
  ...
}
```

Figure 7: *Transformed description of the max/min unit.*

3) Expansion of high-level constructs.
During this step, the maximum and minimum functions are expanded into primitive operations. The expansion of the operations in the first set S_1 are depicted below :

(a) Cluster 1
$tmp = if(in1 > in2) -> in1 || in2;$
$max1 = if(tmp > in3) -> tmp || in3$

(b) Cluster 2
$tmp = if(in1 > in2) -> in2 || in1;$
$min1 = if(tmp > in3) -> in3 || tmp$

4) Definition of the lowly programmable data-paths.
Starting from the 2 expanded clusters, the data-path definition tool *MOZART* automatically generates a dedicated lowly programmable data-path. The result is a structural description of the ASU shown in figure 9.

5) Scheduling of the signal flow graph.
Scheduling in this case is trivial since only 1 clock cycle is available for calculating the inner j loop. The total number of cycles required equals 512^2.

6) Performance optimization of the ASUs.
The generated data-path is optimized w.r.t. the performance constraint : $f_c = 50Hz * 512^2 = 13MHz$.

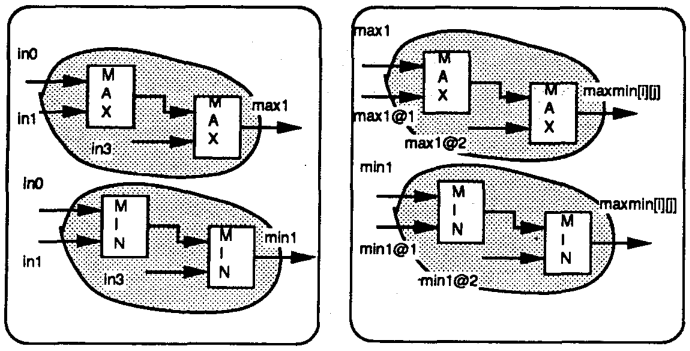


Figure 8: *Obtained clusters for the max/min unit.*

Instead of slow carry-ripple subtractors, faster carry-bypass subtractors are automatically selected by the data-path optimization tool *HANDEL* [9].

3. *Low-level memory and data-path mapping tasks :* Detailed scheduling and fore-ground memory allocation and selection result in the final architecture shown in figure 9.

5 Conclusions.

A novel synthesis environment for high throughput DSP algorithms has been presented. Although scheduling is an important task in a high-level synthesis trajectory, it is only a small part if the synthesis of industrial applications is envisioned. The emphasis is this paper was on the generation of dedicated data-paths which are highly tuned to the signal flow of the DSP algorithm. The advantages of these data-paths are that they are able to solve the computational critical parts in less cycles than is possible on highly programmable data-paths. Furthermore, they are advantageous in terms of interconnect. The different design tasks in the*CATHEDRAL-III*synthesis script have been demonstrated by means of a realistic example.
Besides these advantages, there are some limitations. First of all, the techniques presented are typically oriented towards high throughput applications. Furthermore, some of the tasks are still manual to perform : high-level memory management, and the definition of the clusters in the signal flow graph. Finally, the time complexity of the ASU-definition algorithm is large especially when large sets ($>$ 10) in combination with superoperations containing a large number of operations ($>$ 10).

Acknowledgements
We thank the anonymous reviewers who critically reviewed this paper, and provided us with valuable comments.

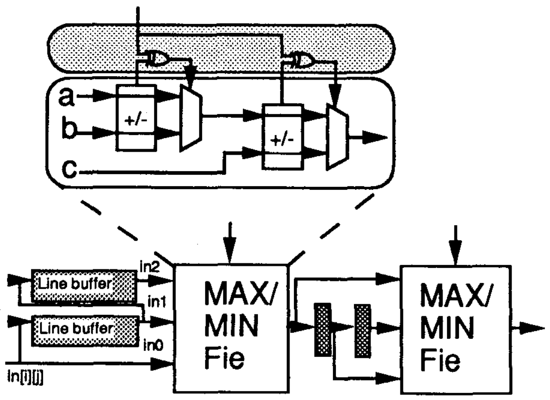


Figure 9: *Final architecture of the max/min unit.*

References

[1] H. De Man, J. Rabaey, P. Six, L. Claesen, "CATHEDRAL-II : A Silicon compiler for Digital Signal Processing", *IEEE Design and Test*, Dec, 1986.

[2] B. Haroun, M. Elmasry, "SPAID : An Architectural Synthesis Tool for DSP Custom Applications", *IEEE J. of Solid State Circuits*, Vol 24, No.2, April, 1989.

[3] S. Devadas and A. R. Newton "Algorithms for Hardware Allocation in Data Path Synthesis", *IEEE Transactions on the Computer Aided Design of Integrated Circuits and Systems*, pp. 768-781, Vol.8 No.7, July 1989.

[4] N. Park, A. Parker, "Sehwa : A Software Package for Synthesis of Pipelines from Behavioural Specifications", *IEEE Transactions on Computer Aided Design*, Vol. 7, No. 3, March, 1988.

[5] N. Park, F. J. Kurdahi "Module Assignment and Interconnect Sharing in Register-Transfer Synthesis of Pipelined Data Paths" *Proc. IEEE International Conference on Computer Aided Design*, pp. 16-19, Santa Clara, Calif., Nov. 1989.

[6] K.S. Hwang, A.E. Casavant, C.Chang, M.A. d'Abreu, "Scheduling and Hardware Sharing in Pipelined Data-paths", *IEEE International Conference on Computer-Aided Design*, Santa Clara, California, Nov. 1989.

[7] C. Chu, M. Potkonjak, M. Thaler, J. Rabaey, "HYPER : An interactive Synthesis Environment for High Performance Real Time Applications", *IEEE Intern. Conference on Computer Design*, Cambridge, 1989.

[8] F.Catthoor, H.De Man, "Application-specific architectural methodologies for high-throughput digital signal and image processing", *IEEE Trans. on Acoustics, Speech and Signal Processing*, pp. 339-349, Vol 38, No 2, Feb 1990.

[9] S. Note, F. Catthoor, G. Goossens, H. De Man, "Combined hardware selection and pipelining in high performance data-path design", *IEEE International Conference on Computer Design*, Sep 1990, Cambridge

[10] S.Note, F.Catthoor, J.Van Meerbergen, H.De Man, "Definition and Assignment of Complex Data-Paths suited for High Throughput Applications", *IEEE International Conference on Computer-Aided Design,* Santa Clara, 6-9 November 1989.

[11] P.N.Hilfinger, "A high-level language and silicon compiler for digital signal processing", *IEEE Custom Integrated Circuits Conf.*, Portland, May 1985.

[12] P.A. Ruetz, "Architectures and Design Techniques for Real Time Image Processing ICs", Phd. Thesis 1986.

[13] W. Geurts, S. Note, F. Catthoor, H. De Man, "Partitioning-based Allocation of Dedicated Data-paths in the Architectural Synthesis for High Throughput Applications", *submitted to the VLSI Conf. 1991*