Memory Accesses Management During High Level Synthesis

Gwenolé Corre, Eric Senn, Pierre Bomel, Nathalie Julien, Eric Martin
LESTER / University of South Brittany
BP92116, 56321 LORIENT cedex, France
firstname.lastname@univ-ubs.fr

ABSTRACT

We introduce a new approach to take into account the memory architecture and the memory mapping in behavioral synthesis. We formalize the memory mapping as a set of constraints for the synthesis, and defined a Memory Constraint Graph and an accessibility criterion to be used in the scheduling step. We present a new strategy for implementing signals (ageing vectors). We formalize the maturing process and explain how it may generate memory conflicts over several iterations of the algorithm. The final Compatibility Graph indicates the set of valid mappings for every signal. Several experiments are performed with our HLS tool GAUT. Our scheduling algorithm exhibits a relatively low complexity that permits to tackle complex designs in a reasonable time.

Categories and Subject Descriptors

B.5 [**RTL Implementation**]: Design Aids

General Terms

Design, Algorithms, Theory, Experimentation

Keywords

Memory aware, Behavioral synthesis

1. INTRODUCTION

Behavioral synthesis, which is the process of generating automatically an RTL design from an algorithmic description, is an important research area in design automation. Many behavioral specifications, especially in digital signal and image processing, use arrays to represent, store and manipulate ever growing amounts of data. The ITRS roadmap indicates that, in 2011, 90 % of the SoC area will be dedicated to the memory [2]. To tackle the complexity of memory design, we consider as essential to take into account memory accesses directly during the behavioral synthesis, assuming that a reasonable trade-off between the design time and the quality of the results is reached. In the context of HLS, several scheduling techniques actually include memory issues. Among them, most try to reduce the memory cost by estimating the needs in terms of number of registers for a given scheduling, but work only with scalars [9]. Some of them really schedule the memory accesses [7]. They include precise temporal models of those accesses, and try to improve performances without considering the possibility of simultaneous accesses which would ease the subsequent task of register and memory allocation. Works in [4] include the memory during HLS, but is dedicated to control intensive applications. In [10], a first scheduling (force directed) is performed on a Data Flow Graph (DFG); the memory accesses are then rescheduled after the selection and memory allocation to reduce the overall memory cost. The complexity of this scheduling algorithm, however, does not allow to target realistic applications in a reasonable time. In [4], memory accesses are represented as multi-cycle operations in a Control and Data Flow Graph (CDFG). Memory vertices are scheduled as operative vertices by considering conflicts among data accesses. This technique is used in some industrial HLS tools that include memory mapping in their design flow (Monet, Behavioral Compiler) [6]. Memory accesses are regarded as Input/Output. The I/O behavior and number of control step are managed in function of the scheduling mode [5]. In practice, the number of nodes in their input specifications must be limited, to obtain a realistic and satisfying architectural solution. This limitation is again mainly due to the complexity of the algorithms which are used for the scheduling.

In this paper, we propose a new and simple technique to take into account the memory mapping in the architectural synthesis. Indeed, our aim is to produce a simple algorithm to achieve the synthesis of even complex designs in a reasonable time. We focus on the definition of a memory mapping file that is used in the synthesis process. We introduce an original scheduling in the synthesis flow, to obtain an optimized RTL design. This scheduling technique is described in section 2 with the formalism to resolve scheduling under memory constraint. The special case of ageing data is discussed in section 3. Experimental results are presented in section 4.

2. MEMORY INTEGRATION

2.1 Memory aware synthesis

We introduce memory synthesis in the standard HLS design flow. A Signal Flow Graph (SFG) is first generated from the algorithmic specification. In the new approach, this SFG is parsed and a memory table is created . This memory table is then completed by the designer who can select the variable implementation (memory or register) and place the variable in the memory hierarchy (which bank). The resulting table is the memory mapping that will be used in the synthesis. In the standard flow, the processing unit is synthesized without any knowledge on the memory mapping. The memory architecture is designed afterward and a lot of optimization opportunities are definitely lost.

The memory mapping file contains information about every data structure in the algorithm (mainly arrays in DSP applications) and its allocation in memory (bank number and physical address). Scalars can also be defined. This memory table represents all data vertices extracted from a SFG. This data distribution can be static or dynamic. In the case of a static placement, the data stay at the same place during the whole execution. If the placement is dynamic, data can be transferred between different levels in the memory hierarchy. Thus, several data can share the same location in the circuit memory. The memory mapping file explicitly describes the data transfers to occur during the algorithm execution. Direct Memory Address (DMA) directives will be added to the code to achieve these transfers. The definition of the memory architecture will be performed in the first step of the overall design flow. To achieve this task, advanced compilers such as Rice HPF compiler, Illinois Polaris or Stanford SUIF could be used [8]. Indeed, these compilers automatically perform data distribution across banks, determine which access goes to which bank, and then schedule to avoid bank conflicts. The Data Transfer and Storage Exploration (DTSE) method from IMEC and the associated tools (ATOMIUM, ADOPT) are also a good mean to determine a convenient data mapping [3].

2.2 Signal Flow Graph

The input of our HLS tool is an algorithmic description that specifies the circuit's functionality at the behavioral level, disregarding any potential implementation solutions. This initial description is compiled in order to obtain an intermediate representation: the Signal Flow Graph (SFG), Fig. 1. A Signal Flow Graph is a directed polar graph $SFG(V,E)$ where the set of vertices $V = \{v0, ..., vn\}$ represents the operations, $v0$ and vn are respectively the source vertex and the sink vertex. The set of edges $E = \{(vi, vj)\}$ represents the dependencies between the operations vertices. The Signal Flow Graph contains $|V| = n+1$ vertices. A vertex represents one of the following operations: arithmetic, logical, data or delay. The difference between a Signal Flow Graph and Data Flow Graph resides in the introduction of delay operators (z^{-k}). These operators are necessary to express the use of data whose value was computed in a preceding iteration of the algorithm. An edge $Ei, j = (vi, vj)$ represents a data dependence between operations vi and vj such as for any iteration of the SFG, operation vi must start its execution before that of vj. For the data dependencies, the execution of vj can start only after the completion of operation vi.

2.3 Memory Constraint Graph

As outlined in section ; all data vertices are extracted from the SFG to construct the memory table. The designer can choose the data to be placed in memory and defines a memory mapping. For every memory in the memory table, we construct a weighted Memory Constraint Graph (MCG). It represents conflicts and scheduling possibilities between all nodes placed in this memory. The MCG is constructed from the SFG and the memory mapping file. We parse the SFG to find the data vertices. The memory table is constructed, where the designer adds mapping information. A Memory Constraint Graph is a cyclic directed polar graph $MCG(V', E', W')$ where $V' = \{v'0, ..., v'n\}$ is the set of data vertices placed in memory. A memory Constraint Graph contains $|V'| = n + 1$ vertices which represent the memory size, in term of memory elements. The set of edges $E' = \{(v'i, v'j)\}$ represents possible consecutive memory accesses, and W' is a function that represents the access delay between two data nodes. W' has only two possible values: Wseq (sequential) for an adjacent memory access in memory, or Wrand (randomize) for a non adjacent memory access. Weight depends on the data placement defined in the memory file. There are as much sub-graphs as memory banks in memory. The memory table gives the number of banks and the address of every data in memory. This permits to construct the MGC. Figure 1 shows a MCG for the LMS filter with two simple port memory banks. The input samples $x(i)$ are placed consecutively in one bank. The filter coefficients $h(i)$ are placed consecutively in one another bank (dotted edges represent edges where W = Wseq).

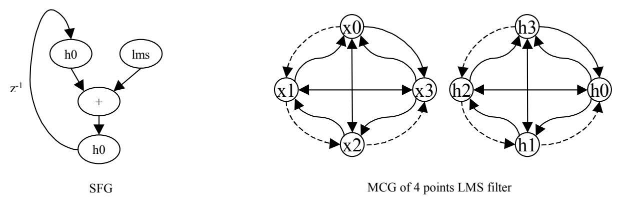


Figure 1: Signal Flow Graph and Memory constraint graph LMS

Memory constraint graphs are used during the scheduling process to determine the accessibility criterion and the time of every memory access.

2.4 Scheduling under memory constraint

The classical list scheduling algorithm relies on heuristics in which ready operations (operations to be scheduled) are listed by priority order. In our tool, an early scheduling is performed. In this scheduling, the priority function depends on the mobility criterion. This mobility is computed, for each cycle, as the difference, in number of cycles, between the current cycle and the operation deadline. Whenever two ready operations need to access the same resource (this is a so called resource conflict), the operation with the lower mobility has the highest priority and is scheduled. The other is postponed. To perform a scheduling under memory constraint, we introduce fictive memory access operators and add an accessibility criterion based on the MCG. A memory has as much access operators as access ports. The memory is declared accessible if one of its fictive memory access operators is idle. Several operations can try to access the same memory in the same cycle; accessibility is used to determine which operations are really executable. The list of ready operations is still organized according to the mobility

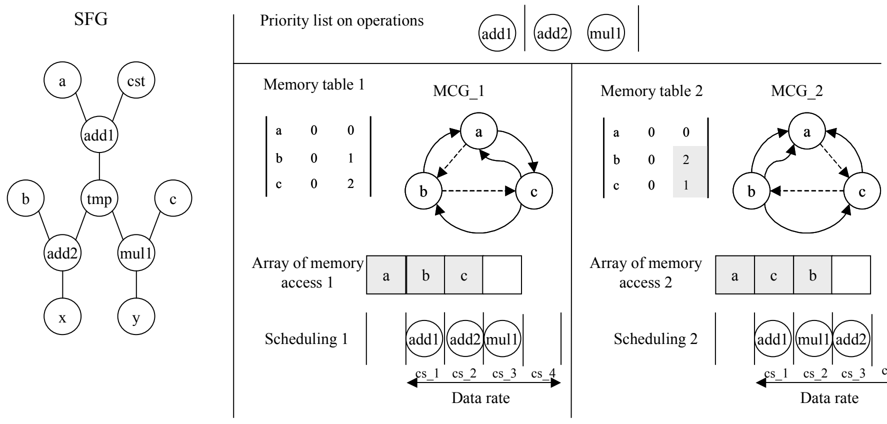


Figure 2: scheduling with MCG

criterion, but all the operations that do not match the accessibility condition are removed from this list. To schedule an operation that involves an access to the memory, we check if the data is not in a busy memory bank. If a memory bank is not available, every operation that needs to access this memory will not be scheduled, no matter its priority level. The MCG is also used to compute the shortest memory access sequence when it's possible.

Our scheduling technique is illustrated in Fig. 2. The memory table is extracted from the SFG. The designer has defined two different memory mappings in memory table 1 and in memory table 2. Data a,b and c are placed at address in bank0. The constant cst is not stored in RAM. Our tool constructs two Memory Constraint Graphs MCG_1 and MCG_2. For mapping 1, the sequential access sequence is $a \rightarrow b \rightarrow c$: it includes two dotted edges (with weight Wseq) $a \rightarrow b$ and $b \rightarrow c$. MCG_2 contains two different dotted edges : $a \rightarrow c$ and $c \rightarrow b$. To deal with the memory bank access conflicts, We define a table of access for each port of a memory bank. In our example, the table has only one line for the single port memory bank0. The table of memory access has Data_rate/Sequential_access_time elements. The value of each element of the table indicates if a fictive memory access operator is idle or not at the current C_step. We use the MCG to produce a scheduling that permits to access the memory in burst mode. If two operations have the same mobility and request to the same memory bank, the operation that is scheduled is the operation that involves an access at an address consecutive with the preceding access. For example, operations add2 and mult1 have the same mobility. At c_step cs_2, they are both executable, the operands (stored in bank0) of add2 and mul1 are respectively data b and data c, the latest data accessed in bank0 is data a. MCG_1 indicates that the access sequence $a \rightarrow b$ is shorter than access $a \rightarrow c$. We schedule add2 at c_step cs_2 to favorize the sequential access. On a contrary, for mapping 2, MCG_2 indicates that mul1 must be scheduled before add2.

3. IMPLEMENTING AGEING VECTORS

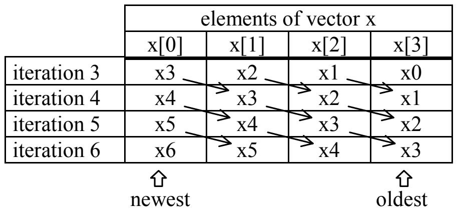
Signals are the input and output flows of the applications. A mono-dimensional signal x is a vector of size n, if n values of x are needed to compute the result. Every cycle, a new value for x ($x[n+1]$) is sampled on the input, and the oldest value of x ($x[0]$) is discarded. We called x an ageing, or maturing, vector or data. Ageing vectors are stored in RAM. A straightforward way to implement, in hardware, the maturing of a vector, is to write its new value always at the same address in memory, at the end of the vector in the case of a 1D signal for instance (that is how Monet works). Obviously, that involves to shift every other values of the signal in the memory to free the place for the new value. This shifting necessitates n reads and n writes in the memory, which is very time and power consuming. In GAUT, the new value is stored at the address of the oldest one in the vector. Only one write is needed. Obviously, the address generation is more difficult in this case, because the addresses of the samples called in the algorithm change from on cycle to the other. The Figure 3 illustrates this difficulty. In the following code a signal x is accessed; it includes $N = 4$ elements.

```
ALGORITHM 1
x(0):=x_input; //new sample of vector x
tmp := x(0);
for i=1 to N-1 loop
   tmp=tmp+x(i);
end loop;
for N-1 to 1 loop
   x(i)=x(i-1); // ageing loop
end loop;
```

The logical address of an element of x ($x[0]$ for instance) changes from an iteration to the other. The logical address of $x[0]$ is that of $x3$ in iteration 3, $x4$ in iteration 4, $x5$ in iteration 5 etc. With GAUT, we make the distinction between physical and logical addresses.The logical address points on a memory element that contains the physical address of the data. The physical address points on the memory element that contains the value of the data. Once determined, the physical address of a data never changes. In our example, for instance, the physical address of data $x3$ from vector x will remains the same as long as $x3$ is alive in the memory.

We have developed a new methodology to resolve the synthesis of our logical address generators. The advantage is a lower latency, since we avoid n reads and writes of the ageing vector, and a resulting lower power consumption. Indeed, the power consumption of a memory increases with the number of accesses.

This methodology is based on an oriented graph that traces the evolution of the logical addresses in a vector dur-



	elements of vector x			
	x[0]	x[1]	x[2]	x[3]
iteration 3	x3	x2	x1	x0
iteration 4	x4	x3	x2	x1
iteration 5	x5	x4	x3	x2
iteration 6	x6	x5	x4	x3
	newest			oldest

Figure 3: The maturing process

ing the execution of one iteration of the algorithm: the *Logical Address Graph* (LAG). The LAG is a couple $LAG = (V, E)$; it is defined for each ageing vector in the algorithm. V is the set of vertices $V = \{v_0, v_1, \cdots, v_{N-1}\}$ where vertex v_i is the i^{th} element of the vector. With x a vector of size N; $card(V) = N$. E is the set of edges $E = \{e_1, e_2, \cdots, e_M\}$ where edge $e = (v_i, v_j)$ links 2 elements v_i and v_j if the j^{th} element of the vector ($x[j]$) is accessed immediately after the i^{th} element of the vector ($x[i]$). $E \subseteq V \times V$. The weighting function f is associated to the LAG; $f : V \times V \rightarrow \mathbb{N}$. For every edge $e = (v_i, v_j)$, f gives the weight f_{ij} with $f_{ij} = (j - i)\%N$. % expresses the modulo. Figure 4 represents the LAG for the preceding example.

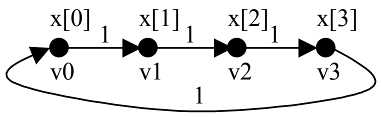


Figure 4: LAG for algorithm 1

The weight f_{ij} is used to calculate the logical address of the next access to vector x. Suppose that 0 is the logical address of $x[0]$. Then $x[1]$ is the next access to x and its logical address is $0 + f_{01} = 1$. The logical address of $x[2]$ is 2 and the logical address of $x[3]$ is 3. The next data to be accessed is $x[0]$. Its address is still $(3+1)\%4 = 0$ in this iteration. However, to calculate the address of $x[0]$ in the next iteration, we ought to take into account the ageing of vector x. In our example, the values in vector x are shifted so that the logical address of element $x[i]$ at the iteration $o+1$, noted $@x[i]^{o+1}$ is the logical address of element $x[i]$ at the iteration o plus 1: $@x[i]^{o+1} = @x[i]^{o} + 1$. In general, we define the *ageing factor* k as the difference between the logical address of element $x[i]$ at the iteration $o+1$ and the logical address of element $x[i]$ at the iteration o. In our example, $k = 1$.

$$k = @x[i]^{o+1} - @x[i]^{o} \tag{1}$$

Eventually, to calculate the logical address of $x[0]$, we add (modulo N), to the logical address of $x[3]$ in the preceding iteration, the weight f_{30} and the *ageing factor* k so that $@x[0] = (@x[3] + f_{30} + k)\%N$. More generally, if $x[i]$ is the last element of x accessed in iteration o and $x[j]$ is the first element of x accessed in iteration $o+1$, and with N the size of x:

$$@x[j]^{o+1} = (@x[i]^{o} + f_{ji} + k)\%N \tag{2}$$

Consider the algorithm below. This algorithm was synthesized with the following mapping: $x[0]$ and $x[1]$ are in a memory bank, $x[2]$, $x[3]$ and $x[4]$ are in another bank. The relation with the logical addresses is determined for the first iteration. So $@x[0]$ (= 0) and $@x[1]$ (= 1) are in the first bank, $@x[2]$ (= 2), $@x[3]$ (= 3) and $@x[4]$ (= 4) are in the second.

```
ALGORITHM 2
x(0):=x_input; //new sample of vector x
tmp:=x(0);
tmp:=tmp+ x(1);
tmp1:=x(2);
tmp1:=tmp1+x(3);
tmp1:=tmp1+x(4);
for N-1 to 1 loop
   x(i)=x(i-1); // ageing loop
end loop;
```

The LAG for vector x is represented Figure 5. The chronogram of accesses is presented figure 6.

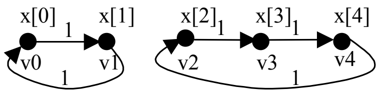


Figure 5: LAG for algorithm 2

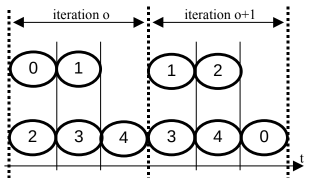


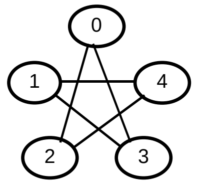
Figure 6: Chronogram of accesses

In iteration o, several concurrent accesses to the memory appear: $@x[0] = 0$ with $@x[2] = 2$, and $@x[1] = 1$ with $@x[3] = 3$. These parallel accesses do not generate conflict for involved data are in distinct memory banks. In the next iteration however, the logical addresses for $x[1]$ and $x[3]$ are respectively 2 and 4. A memory conflict is generated since these two logical addresses are mapped in the same memory bank. The concurrent accesses are computed for N successive iterations of the algorithm to obtain the concurrent accesses table (see table 1).

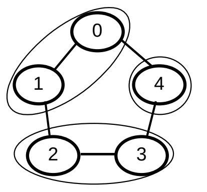
Table 1: Logical addresses evolution and concurrent accesses table

iteration	x[0]	x[1]	x[2]	x[3]	x[4]	concurrent accesses table
o	0	1	2	3	4	(0,2) / (1,3)
o+1	1	2	3	4	0	(1,3) / (2,4)
o+2	2	3	4	0	1	(2,4) / (3,0)
o+3	3	4	0	1	2	(3,0) / (4,1)
o+4	4	0	1	2	3	(4,1) / (0,2)
o+5	0	1	2	3	4	(0,2) / (1,3)
	logical addresses					

The *set of concurrent accesses* (SCA) is the set of all the concurrent accesses in the concurrent accesses table. In our example, $SCA = \{(0,2),(1,3),(2,4),(3,0),(4,1)\}$. A *Concurrent Accesses Graph* (CAG) is constructed from this set of concurrent accesses. A CAG is a couple $CAG = (L, A)$. L is the set of vertices $L = \{l_0, l_1, \cdots, l_{N-1}\}$ where vertex l_i is the logical address of the i^{th} element of the vector in the first iteration. With x a vector of size N; $card(L) = N$. A is the set of edges $A = \{a_1, a_2, \cdots, a_M\}$ where edge $a = (l_i, l_j)$ links 2 elements l_i and l_j if the couple (l_i,l_j) is included in the set of concurrent accesses. $A \subseteq L \times L$. Figure 7(a) gives the conflict graph for our example.



(a) Concurrent Accesses Graph



(b) Compatibility Graph

Figure 7: CAG and CG

In this case, the synthesis is not possible. GAUT issues a message to indicate that the data mapping is not valid. To help in determining a valid data mapping, a *Compatibility Graph* (CG) is constructed. The CG is orthogonal to the former Conflict Graph (Figure 7(b)). The minimum number of memory banks is easily computed from the Compatibility Graph. In our example, the minimum number of memory banks is 3. A possible mapping is to place $x[0]$ and $x[1]$ in a first bank, $x[2]$ and $x[3]$ in a second bank, and $x[4]$ in a third bank. It is remarkable that these results actually depend on the scheduling, and therefore on the timing constraint provided to the tool. With a different timing constraint, the conflict and compatibility graphs change, as well as the set of valid data mappings.

Similar results are obtained when pipelined architectures are synthesized. The chronogram of accesses for algorithm 1 is presented on Figure 8. When the architecture is pipelined, this chronogram is modified as shown on Figure 9.

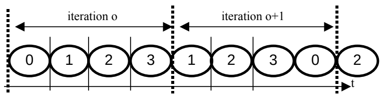


Figure 8: Non-pipelined architecture

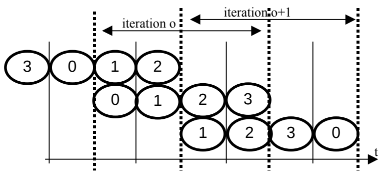
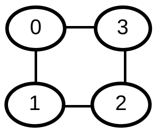
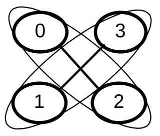


Figure 9: Pipelined architecture

The situation is similar to the situation with the algorithm 2: concurrent accesses appear and a concurrent accesses table is determined. The difference is that the conflicts arise between logical addresses that are calculated over several successive iterations (2 in this example). $@x[2]^o$ is in concurrence with $@x[0]^{o+1}$, and $x[3]^o$ is in concurrence with $@x[1]^{o+1}$. The set of concurrent accesses $SCA = \{(0,1),(1,2),(2,3),(3,0)\}$. The CAG and CG are computed from this SCA (Figure 10). The data mapping is verified. The minimum number of banks is 2, and the only valid mapping with 2 banks is to place $x[0]$ and $x[2]$ in a bank, and $x[1]$ and $x[3]$ in another bank.



(a) CAG



(b) CG

Figure 10: CAG and CG

4. GAUT VS MONET

Several syntheses were performed, both with GAUT and the industrial behavioral synthesis tool Monet. We chose the elliptic and the Kalman filters which are the biggest applications in the HLSynth'92 benchmarks [1], and two classical digital algorithms: a FIR filter and an echo cancellation algorithm, the LMS. Table 2, indicates the synthesis time in seconds and the architecture's latency in number of cycles (the same real-time constraint was given to the tools, the clock cycle is 10ns). Required hardware resources are also indicated: the number of registers (Reg), of multiplexers (Mux), demultiplexers (Demux), of glue logic elements (which are tri-states in GAUT), and the number of RAM and ROM memories. The two last columns give the number of read and write in those memories. Single port SRAM were used to store data. Syntheses were executed on SUN Blade 2000 workstations.

Hardware resources are always lower in architectures synthesized with GAUT, although the same number of arithmetic operators is needed. The latency, which is the delay between the input of the first data and the first result on the output, is also lower with GAUT. A ROM is needed with GAUT for the FIR filter, since GAUT stores every static coefficient in ROM. Those coefficients are wired with Monet. Dynamic coefficients, whose value is changed during the execution of the algorithm, which is the case for an adaptative filtering like the LMS, are stored in RAM, together with signals (ageing vectors). The advantages of our approach appear clearly here: the latency is lower with GAUT since we avoid the n reads and writes of the ageing vector performed with Monet. As a result, the power consumption decreases.

The synthesis time, together with the reduction of hardware resources and memory accesses, exhibit the efficiency of our scheduling technique. In fact, the difference between the synthesis time with GAUT and with a behavioral synthesizer like Monet increases with the complexity of the application. We have measured the synthesis times for the FIR and the LMS filters, with an increasing complexity. Table 3 presents the results for the FIR for 32, 128, 512, and 1024 points. Table 4 presents the results for the LMS filter for the same increasing complexities. It can be observed that, even if the difference between the synthesis time with GAUT

Table 2: GAUT vs Monet

		Synth time	Lat Nb_cycle	Reg	Mux	Demux	Tri	Glue	RAM	ROM	Nb read	Nb write
elliptic	Monet	1s	20	19	16	15	–	27	–	–	–	–
	Gaut	1s	20	12	6	9	24	–	–	–	–	–
Kalman	Monet	1s	60	36	12	20	–	34	–	–	–	–
	Gaut	1s	60	14	11	10	29	–	–	–	–	–
FIR 16	Monet	2s	48	4	6	2	–	7	1	–	32	16
	Gaut	1.4s	19	4	2	1	1	–	1	1	32	1
LMS 32	Monet	6s	132	38	28	18	–	25	2	–	128	64
	Gaut	1.4s	100	19	3	3	23	–	2	–	128	33

and Monet is relatively small for small designs, it becomes enormous when the design's complexity increases. Indeed, it becomes hours, then days or weeks for the FIR 1024 and the LMS 512 and 1024. In fact, every memory access is a node to be schedule in Monet, and the scheduling algorithm has a strong complexity. The difference in latency is comparatively stable: the latency with Monet varies from about 2 to 3 times the latency with GAUT.

Table 3: Synthesis of the FIR filter

FIR	Tool	cycles	Reads	Writes	Time
32	Monet	96	64	32	3s
	Gaut	35	64	1	1.5s
128	Monet	384	256	128	45s
	Gaut	131	256	1	2s
512	Monet	1536	1024	512	7h17mn
	Gaut	512	1024	1	4.9s
1024	Monet	3072	2048	1024	... days
	Gaut	1027	2048	1	9s

Table 4: Synthesis of the LMS filter

LMS	Tool	cycles	Reads	Writes	Time
32	Monet	132	128	64	6s
	Gaut	100	128	33	1.4s
128	Monet	516	512	256	7mn30s
	Gaut	388	512	129	2.6s
512	Monet	2052	2048	1027	... days
	Gaut	1540	2048	513	9.6
1024	Monet	4010	4096	2048	... weeks
	Gaut	3076	4096	1025	64

5. CONCLUSION

In this paper, we present two recent improvements to our High-Level Synthesis tool GAUT. We first define the memory mapping constraint and include it in the synthesis design flow. We introduce the Memory Constraint Graph, and an accessibility criterion to enhance the scheduling algorithm. We show that a peculiar attention must be paid to signals, or ageing vectors. We formalize the maturing process and explain how it may generate memory conflicts over several iterations of the algorithm. We define the Logical Accesses Graph, and the Concurrent Accesses Table, which are used to construct the Concurrent Accesses Graph, and the Compatibility Graph. The Compatibility Graph indicates the minimum number of memory banks for the scheduling, and helps in finding a valid mapping for signals.

Several experiments were made, to explore the efficiency of our approach. The comparison with an industrial behavioral synthesis tool exhibits several advantages for GAUT.

In the future, the scheduling step will be enhanced with an anticipated read model for the data, which should allow to speedup the processing unit. The presented strategy for implementing ageing vectors will be reversed, in order to automatize the determination of the memory mapping for this type of data.

6. REFERENCES

[1] HLSynth'92 benchmark information. http://www.cbl.ncsu.edu/CBL_Docs/hls92.html

[2] ITRS homepage. http://public.itrs.net/

[3] F. Catthoor, K. Danckaert, C. Kulkarni, and T. Omns. *Data Transfer and Storage (DTS) architecture issues and exploration in multimedia processors*. Marcel Dekker Inc., NewYork, 2000.

[4] P. Ellervee. *High-Level Synthesis of Control and Memory Intensive Applications*. PhD thesis, Royal Institut of Technology, Jan. 2000.

[5] D. Knapp, T. Lyand, et al. Behavioral synthesis methodology for HDL-based specification and validation. In *Proc. Design Automation Conference DAC'95*, June 1995.

[6] H. Ly, D. Knapp, R. Miller, and D. McMillen. Scheduling using behavioral templates. In *Proc. Design Automation Conference DAC'95*, pages 101–106, June 1995.

[7] A. Nicolau and S. Novack. Trailblazing a hierarchical approach to percolation scheduling. In *Proc. ICPP'93*, pages 120–124, 1993.

[8] P. Panda et al. Data and memory optimization techniques for embedded systems. *Transactions on Design Automation of Electronic Systems*, 6(2):149–206, 2001.

[9] R. Saied and C. Chakrabarti. Scheduling for minimizing the number of memory accesses in low power applications. In *Proc. VLSI Signal Processing*, pages 169–178, Oct. 1996.

[10] J. Seo, T. Kim, and P. Panda. An integrated algorithm for memory allocation and assignment in high-level synthesis. In *Proc. Design Automation Conference DAC'01*, pages 608–611, June 2001.