High Level Ageing Vectors Management for Data Intensive Applications

Gwenolé Corre, Eric Senn, Nathalie Julien, Eric Martin

▶ **To cite this version:**

Gwenolé Corre, Eric Senn, Nathalie Julien, Eric Martin. High Level Ageing Vectors Management for Data Intensive Applications. 2005. ⟨hal-00077305⟩

HAL Id: hal-00077305

https://hal.science/hal-00077305v1

Submitted on 30 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

High Level Ageing Vectors Management for Data Intensive Applications

Gwenolé Corre, Eric Senn, Nathalie Julien and Eric Martin
LESTER / University of South Brittany
BP92116, 56321 Lorient cedex, France
Email: gwenole.corre@univ-ubs.fr

***Abstract*— We introduce a new approach to take into account the memory architecture and the memory mapping in behavioral synthesis. 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 strategy exhibits a relatively low complexity memory architecture for ageing vectors that permits to tackle complex designs for data intensive applications.**

I. 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 [1]. Applications are indeed becoming more and more complex, and memory will take a more and more important place in future signal processing systems. This place is vital, strategic, for memory now appears as a terrific bottleneck in real-time systems. Indeed, performances are highly dependent on the memory architecture (hierarchy, number of banks) together with the way data are placed and transferred. 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 [2].In [3], 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 management in their design flow (Monet, Behavioral Compiler) [4]. 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 ageing vectors in the architectural synthesis. The definition of ageing data and their implementation are discussed in section II. Experimental results are presented in section III.

II. 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 1 illustrates this difficulty. In the following code a signal x is accessed; it includes $N = 4$ elements.

```
ALGORITHM 1
x(0):=x input;
tmp := x(0);
for (i=1;i=N-1;1++) tmp=tmp+x(i);
for (i=N-1 ; i= 1 ;i--) x(i)=x(i-1);
```

The logical address of an element of x ($x[0]$ for instance) changes from an iteration to the other with x_i the i^{th} value of signal x . 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.

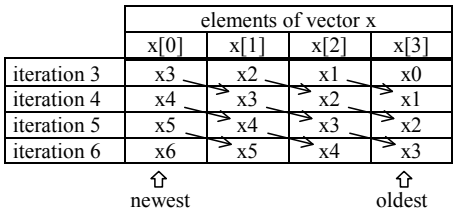


Fig. 1. The maturing process

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 during 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 2 represents the LAG for the preceding example.

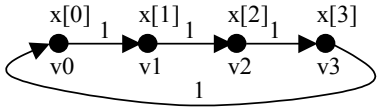


Fig. 2. LAG for algorithm 1

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, $k = 1$.

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

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:

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

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;
tmp=x(0);
tmp=tmp+ x(1);
tmp1=x(2);
tmp1=tmp1+x(3);
tmp1=tmp1+x(4);
for (i=N-1 ; i= 1 ;i--) x(i)=x(i-1);
```

The LAG for vector x is represented Figure 3. The chronogram of accesses is presented figure 4. We indicate in the circle,the logical address of the vector to be fetched.

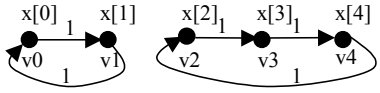


Fig. 3. LAG for algorithm 2

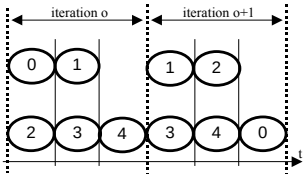


Fig. 4. Chronogram of accesses

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 I).

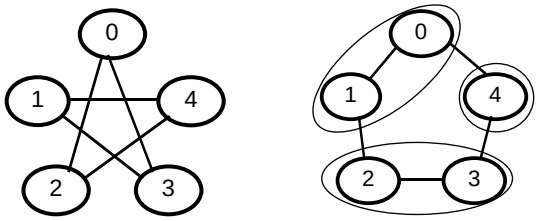
The *set of concurrent accesses* (SCA) is the set of all the concurrent accesses in the concurrent accesses table. In

TABLE I

LOGICAL ADDRESSES EVOLUTION AND CONCURRENT ACCESSES TABLE

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

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 5(a) gives the conflict graph for our example.



(a) Concurrent Accesses Graph (b) Compatibility Graph

Fig. 5. 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 5(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 6. When the architecture is pipelined, this chronogram is modified as shown on Figure 7.

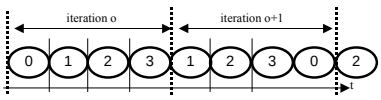


Fig. 6. Non-pipelined architecture

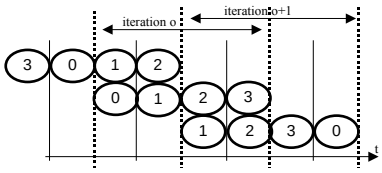
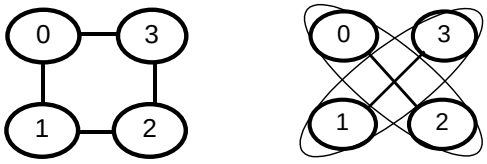


Fig. 7. Pipelined architecture

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 8). 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

Fig. 8. CAG and CG

III. GAUT VS INDUSTRIAL HLS TOOLS

Several syntheses were performed, both with GAUT and the industrial behavioral synthesis tools Monet from Mentor Graphic and Behavioral Compiler from Synopsys. We chose the elliptic and the Kalman filters which are the biggest applications in the HLSynth'92 benchmarks [6], and two classical digital algorithms: a FIR filter and an echo cancellation algorithm, the LMS. Table II, 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 and BC. 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

TABLE II

GAUT VS INDUSTRIAL TOOL

		Synth time	Lat (Nb_cycle)	Reg	Mux	Demux	Tri	Glue	RAM	ROM	Nb read	Nb write
elliptic	Monet	1s	20	19	16	15	–	27	–	–	–	–
	BC	1s	20	18	14	10	–	27	–	–	–	–
	Gaut	1s	20	12	6	9	24	–	–	–	–	–
Kalman	Monet	1s	60	36	12	20	–	34	–	–	–	–
	BC	1s	60	24	12	16	–	32	–	–	–	–
	Gaut	1s	60	14	11	10	29	–	–	–	–	–
FIR 16	Monet	2s	48	4	6	2	–	7	1	–	32	16
	BC	2s	35	4	4	2	–	6	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
	BC	4s	132	32	24	14	–	22	2	–	128	64
	Gaut	1.4s	100	19	3	3	23	–	2	–	128	33

(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 and BC. As a result, the power consumption decreases. Indeed, the power consumption of a memory increases with the number of accesses. 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 HLS tools like Monet and BC 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 III presents the results for the LMS for 32, 128, 512, and 1024 points. It can be observed that, even if the difference between the synthesis time with GAUT and industrial tools 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 LMS 512 and 1024. In fact, every memory access is a node to be schedule in Monet and BC, and the scheduling algorithm has a strong complexity. The difference in latency is comparatively stable: the latency with Monet and BC varies from about 2 to 3 times the latency with GAUT.

IV. CONCLUSION

In this paper, we present two recent improvements to our High-Level Synthesis tool GAUT. Our goal is to take into account the memory architecture and the memory mapping in the synthesis process. 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.

TABLE III

SYNTHESIS OF THE LMS FILTER

LMS	Tool	cycles	Reads	Writes	Time
32	Monet	132	128	64	6s
	BC	132	128	64	4s
	Gaut	100	128	33	1.4s
128	Monet	516	512	256	7mn30s
	BC	132	128	64	5mn14s
	Gaut	388	512	129	2.6s
512	Monet	2052	2048	1027	... days
	BC	132	128	64	hours
	Gaut	1540	2048	513	9.6
1024	Monet	4010	4096	2048	... weeks
	BC	132	128	64	days
	Gaut	3076	4096	1025	64

Several experiments were made, to explore the efficiency of our approach. The comparison with industrial behavioral synthesis tools 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.

REFERENCES

[1] ITRS homepage. [Online]. Available: http://public.itrs.net/

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

[3] P. Ellervee, "High-level synthesis of control and memory intensive applications," Ph.D. dissertation, Royal Institut of Technology, Jan. 2000.

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

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