A Complete Data Scheduler for Multi-Context Reconfigurable Architectures

M. Sanchez-Elez, M. Fernandez, R. Maestre, R. Hermida, N. Bagherzadeh[§], F. J. Kurdahi[§]

Departamento de Arquitectura de Computadores y Automatica
Universidad Complutense – 28040 Madrid, SPAIN
e-mail: marcos@fis.ucm.es

[§]Department of Electrical and Computer Engineering
University of California, Irvine – CA 92697, USA

Abstract:
A new technique is presented in this paper to improve the efficiency of data scheduling for multi-context reconfigurable architectures targeting multimedia and DSP applications. The main goal is to improve the applications execution time minimizing external memory transfers. Some amount of on-chip data storage is assumed to be available in the reconfigurable architecture. Therefore the Complete Data Scheduler tries to optimally exploit this storage, saving data and result transfers between on-chip and external memories. In order to do this, specific algorithms for data placement and replacement have been designed. We also show that a suitable data scheduling could decrease the number of transfers required to implement the dynamic reconfiguration of the system.

1. Introduction.

Reconfigurable computing represents an intermediate approach between the extremes of ASICs (Application Specific Integrated Circuits) and general-purpose processors. Reconfigurable systems combine a reconfigurable hardware unit with a software programmable processor. They are an alternative for implementing a wide range of computationally intensive applications such as DSP or multimedia.

FPGAs [1] are the most common devices used for reconfigurable computing. However, dynamic reconfiguration [2] has emerged as a particular attractive technique for minimizing the reconfiguration time, which has a negative effect on FPGAs performance.

An example of dynamic reconfiguration is multi-context architectures, which may store a set of different configurations for the entire reconfigurable chip (contexts) in an internal memory. When a new configuration is needed, it is downloaded from this internal memory, obviously, this operation is faster than the reconfiguration from the external memory.

Previous work [3][4][5] discussed scheduling for multi-context architectures, in particular with MorphoSys [6] as the target architecture. It has been developed for applications with a considerable amount of potential parallelism such as multimedia and DSP applications, which are typically composed of a group of macro-tasks. We use the term kernel to refer to one of this macro-task. At the abstraction level on which we are working a kernel is characterized by its contexts, as well as, its input and output data.

In [7] is proposed a kernel scheduling technique to generate a kernel sequence that estimates the execution time through a tentative context and data schedules. Context scheduling is further refined in [4] and its goal is to minimize the number of context loads that do not overlap with computation. Data scheduling was dealt with in [5] wherein is proposed a method to minimize the data size stored into the internal memory. It allows storing data for the execution of several consecutive iterations of the same cluster, avoiding context reloads. But this first scheduling does not minimize data transfers.

Multimedia and DSP applications deal with a large amount of data, which have to be stored and transferred. The minimization of these transfers from/to external memory achieves an important reduction in the overall execution time. However, this data scheduling for reconfigurable systems has not been discussed in detail by other authors. In [8] a data scheduler for dynamic architectures is proposed, though it does not optimize memory management. A method to minimize memory traffic is suggested in [9], but it does not dealt with memory allocation in detail. From a specific point of view in [10] is proposed a data scheduler for MPEG2. However, most effort on memory organization has been spent on cache organization for general-purpose computer [11] and not on more custom memory organizations, as needed e.g. multimedia and DSP applications.

The Complete Data Scheduler reduces memory transfers minimizing data and context transfers. Moreover, a specific algorithm for data allocation has been designed to optimize memory usage.

The paper begins with a briefly overview of MorphoSys and its compilation framework. Section 3 describes the previous work. Section 4 analyzes data management among

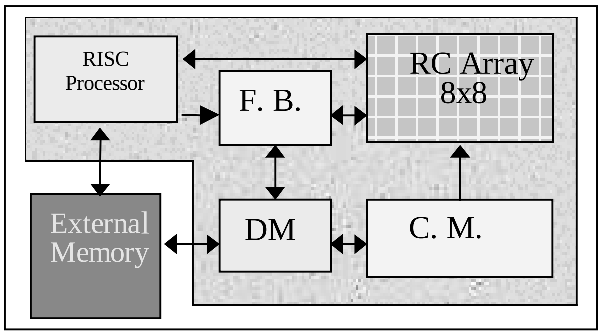

Figure 1. MorphoSys Chip (M1)

clusters to reduce data transfers. An allocation algorithm has been designed, as shown in Section 5. Experimental results are presented in Section 6. Finally, we present some conclusion from our research in Section 7.

2. Architecture and framework overview.

This section briefly describes the target system M1 (first implementation of MorphoSys). We also present the development framework that integrates the data scheduler with other compilation task.

MorphoSys (Fig. 1) is composed of an 8x8 array of reconfigurable cells (RC) with an organization that helps exploit the inherent parallelism of an application for greater throughput. Its functionality and interconnection network are configured through 32-bit context words, which are stored in a context memory (CM). The frame buffer (FB) serves as a data cache for the RC Array. This buffer has two sets to enable overlapping of computation with data transfers. Data from one set is used for current computation, while the other set stores results in the external memory and loads data for the next round of computation. The Complete Data Scheduler reduces this data and results transfers. The DMA controller establishes the bridge that connects the external memory the FB or the CM. Thus simultaneous transfers of data and contexts are not possible. MorphoSys operation is controlled by a RISC processor.

The architecture imposes certain constraints that have to be considered by the scheduling.

An overview of the proposed compilation framework is shown in figure 2. The application code is written in terms of kernels that are available in a kernel library. The kernel programming is equivalent to specifying the mapping of computation to the target architecture, and is done only once.

The information extractor generates the information needed by the compilation tasks that follow it, including kernel execution time, data reuse among kernels, as well as, data size and number of contexts for each kernel.

The kernel scheduler [7] explores the design space to find a sequence of kernels that minimizes the execution time. It decides which is the best sequence of kernels and performs

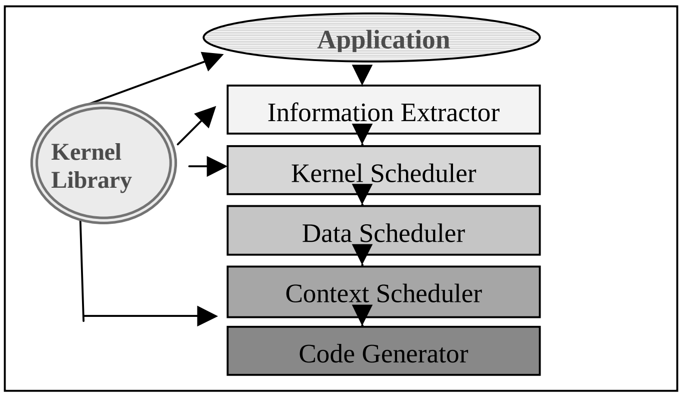

Figure 2. Compilation Framework

clusters. The term cluster is used here to refer to a set of kernels that is assigned to the same FB set and whose components are consecutively executed, e. g. if an application is composed of kernels k_1, k_2, k_3, k_4 and k_5, the kernel scheduler could assign k_1 and k_2 to one FB set, and k_3, k_4 and k_5 to the other set. This implies the existence of two clusters $Cl_1=\{k_1, k_2\}$ and $Cl_2=\{k_3, k_4, k_5\}$. While the first cluster is being executed using data of one FB set, the contexts and data of the other cluster kernels are being transferred to CM and to the other FB set respectively.

The kernel scheduler generates one kernel sequence that minimizes the overall execution time, estimating data and contexts transfers. The context and data schedulers specify when and how each transfer is performed. Their goal is to optimize performance.

3. Data Management within cluster.

We proposed in [5] a methodology to perform data scheduling within a cluster improving FB usage. It reduces data and contexts loads.

Before cluster execution begins, its external data have to be transferred to the FB. Furthermore, the kernel results, obtained after its execution, are stored into the FB. We consider application such that data and result sizes are known before cluster execution, which is the typical case for a wide range of multimedia applications. The data scheduler can calculate the maximum data and result sizes that have to be stored into one FB set to perform cluster execution.

However, many external data, as well as, intermediate results (kernel results that are used as data by kernels executed later) are not used by all kernels of the cluster. The data scheduler takes this fact into account. It replaces the external data or intermediate results that are not going to be used as input data by kernels executed later, with new intermediate and final results. Therefore it reduces the maximum data and result sizes required to cluster execution.

Let $C_C = \{k_1, k_2, \ldots, k_n\}$ be a cluster, then $DS(C_C)$ stands for the maximum data size of cluster c.

$$DS(C_c) = \underset{i\in\{1,\ldots,n\}}{MAX}\left[\sum_{j=i}^{n} d_j + \sum_{j=1}^{i}\left(rout_j + \sum_{t=i}^{n} r_{jt}\right)\right]$$

d_j : size of the input data for kernel kj except those shared with kernel executed later $\{k_{j+1},\ldots,k_n\}$

rout $_j$: size of the result of kj that will be used as data by kernels of clusters executed later.

r_{jt} : size of the intermediate result of k_j, which are data for k_t and not for any kernel executed after k_t.

Multimedia applications, such as DSP or MPEG, are composed of a sequence of kernels that are consecutively executed over a part of the input data, until all the data are processed. If the kernels of the application may be executed 'n' times to process the total amount of data, their contexts may be loaded to CM 'n' times. However sometimes loop-fission may be applied (Figure 3), so that every kernel is executed RF times before executing the next one. The number of consecutive executions of one kernel RF (Context Reuse Factor) is limited by the internal memory size. This data scheduling maximizes free space in the FB, which is used to store cluster data required for RF consecutive iterations. In this case their contexts are only loaded n/RF times, so reducing context reloading and minimizing execution time.

However this data scheduler does not minimize data transfers from/to external memory. The data scheduler suggested in this paper, the Complete Data Scheduler, minimizes these transfers as next section shows. Moreover, a specific data placement policy that tries to simplify accesses to the FB has been developed.

4. Data management among clusters.

The problem could be defined as: 'Given an ordered set of clusters and known their data and result sizes, find the data scheduling that minimizes execution time reducing memory transfers'.

In this section we present the Complete Data Scheduler which minimizes memory transfers.

In order to reduce context transfers it achieves the highest common RF value, to all clusters, allowed by the internal memory size, as was explained in section 3. As a consequence, kernel contexts are reused RF times, so minimizing context transfers. Moreover, The Complete Data Scheduler takes into account that many external data and intermediate results are shared among clusters. It chooses which of these external data and intermediate results have to be kept in the FB to avoid transfers.

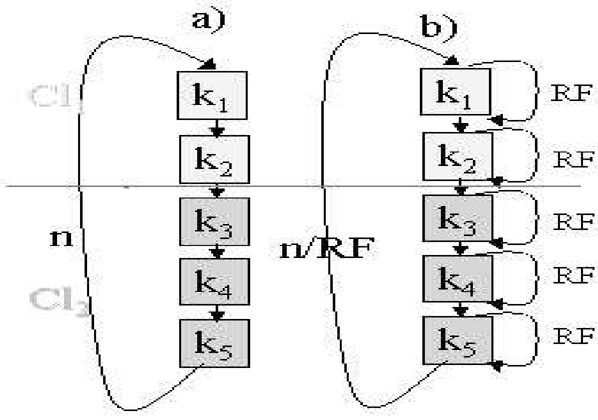

Figure 3: a) Kernel scheduling graph, b) Kernel scheduling graph when is applied loop-fission

The Complete Data Scheduler finds the shared data and the shared results among clusters. For these cases, $D_{i\ldots j}$ stands for the size of the data shared among clusters $\{C_i, \ldots, C_j\}$ which are assigned to the same FB set. And $R_{i,j\ldots k}$ (shared results) stands for the size of cluster i results that are input data for clusters $\{C_j, \ldots, C_k\}$ which are assigned to the same FB set.

The Complete Data Scheduler chooses the shared data or results to be kept into FB according to a factor TF (time factor), which reflects the time saving gained from keeping these shared data or results:

$$TF(D_{i\ldots j}) = \frac{D_{i\ldots j}\cdot(N-1)}{TDS}$$

$$TF(R_{i,j\ldots k}) = \frac{R_{i,j\ldots k}\cdot(N+1)}{TDS}$$

N: number of clusters that use as input data these shared data or result.

TDS: Total data and result sizes.

TF expressions for shared data and for shared results are different. This happens because in the case of shared data, these have to be loaded once from the external memory, therefore, 'N-1' is the number of transfers avoided. However, in the case of shared results they do not have to be stored into the external memory if they are kept into the FB. And they do not have to be reloaded when the clusters that use these shared results as input data are executed. Therefore, 'N+1' is the number of transfers avoided.

The Complete Data Scheduler shorts the shared data and results according with TF. It starts checking that $DS(C_c) \leq$ FBS (frame buffer set size) for all clusters assigned to that FB set for shared data or results with the highest TF. Scheduling continues with shared data or results with less TF. If $DS(C_c)$>FBS for some shared data or results, these are not kept. The Complete Data Scheduler keeps the highest possible amount of shared data or results that minimizes data reloads or unnecessary results stores into the external memory, so reducing data transfers and minimizing execution time.

5. Data and results allocation algorithm.

Data and results reuse within a cluster or among clusters makes data and results placement complex. An allocation algorithm has been developed to minimize fragmentation, as well as, to promote regularity to simplify data and results accesses when kernels are iteratively executed.

This algorithm, used by the Complete Data Scheduler, improves Data Scheduler allocation algorithm [5] reducing fragmentation, and allowing results reuse among clusters.

Data and result sizes are known at compilation time, and kernel input data and result sizes for different iterations are the same. For these reasons, data scheduling has to be performed only once.

As FB is not a large memory and as data and result sizes are similar, the chosen allocation method is first-fit. It keeps track of which parts are free through a linear list of all free blocks (***FB_list***).

Before the beginning of cluster c (C_c) execution, its input data, which have not been stored into FB yet, have to be loaded from the external memory. Some input data of C_c are shared with next clusters. As these data are going to remain longer in the FB than others input data, they are placed first to minimize fragmentation. The procedure ***allocate_shared_data(c,v,RF)*** places the data shared between cluster c and cluster v following the first-fit algorithm from upper free addresses RF times (Figure 4).

After, the Complete Data Scheduler places the C_c input data, which are not shared with other clusters. The procedure ***allocate_kernel_data(c,k,RF)*** performs the kernel k of cluster c data allocation following first-fit from upper free addresses RF times.

The execution of kernel k of cluster c produces different results that have to be placed.

The procedure ***allocate_shared_result(c,k,v,iter)*** places the kernel k of cluster c results shared with cluster v for iteration *iter* following first-fit algorithm from upper free addresses. Because these results are data for next clusters.

The final results (results that have to be transferred in the external memory) are placed by ***allocate_final_results(c,k,iter)***. It performs the kernel k of cluster c final results allocation following first-fit algorithm from lower free addresses for iteration ***iter***.

The procedure ***intermediate_result_allocation(c,k,t,iter)*** performs the kernel k of cluster c results allocation following first-fit from lower addresses. It places the results of kernel k, which are used as input data for kernel t.

When kernel k execution finalizes, the Complete Data Scheduler releases the space occupied by data and results,

```
For c = first cluster to last cluster do {
    For v = last cluster down to c+2 do {
      allocated_shared_data(c,v,RF)
   }
   For k = last kernel down to  first kernel do {
     allocate_kernel_data(c,k,RF)
   }
   For iter = 1 to  RF do {
     For k = first kernel to last kernel do {
       For all the shared result of kernel k do {
         For v = last cluster down to  c+2 do {
            allocate_shared_results(c,k,v,iter)
         }
      }
       else for all the final result of kernel k do {
         allocate_final_result(c,k,iter)
      }
      else {
         for t = last kernel down to k+1 do {
           allocate_intermediate_results(c,k,t,iter)
         }
      }
     release(c,k,iter)
     }
   }
}
```

Figure 4: Allocation Algorithm

a	b	c	d	e	f	g
D_{13}	D_{13}	D_{13}	D_{13}	▨	$\mathbf{R_{3,5}}$	$R_{3,5}$
D_{13}	D_{13}	D_{13}	▨	▨	$\mathbf{R_{3,5}}$	$R_{3,5}$
▨	$\mathbf{D_{37}}$	D_{37}	D_{37}	D_{37}	D_{37}	D_{37}
▨	$\mathbf{D_{37}}$	D_{37}	D_{37}	D_{37}	D_{37}	D_{37}
▨	$\mathbf{d_2}$	d_2	d_2	*d_2*	***R_{out}*** ***R_{out}***	▨
▨	$\mathbf{d_2}$	d_2	d_2	$\mathbf{r_{23}}$	*r_{23}*	▨
▨	$\mathbf{d_1}$	d_1	*d_1*	r_{23}	▨	▨
▨	$\mathbf{d_1}$	*d_1*	$\mathbf{r_{13}}$	r_{13}	*r_{13}*	▨
▨	▨	$\mathbf{r_{13}}$	r_{13}	r_{13}	▨	▨

***a**) FB before cluster 3 execution**.** **b**) FB after load cluster 3 input data. **c**) FB after first execution of kernel 1. **d**) FB after second execution of kernel 1**.** **e**) FB after second execution of kernel 2**.** **f**) FB after second execution of kernel 3. **g**) FB before cluster 5 execution.*

Figure 5: Example of FB allocation. Execution of 3 kernel of cluster 3 with RF=2

which are not going to be used by next kernels of cluster c or next clusters. The procedure ***releases(c,k,iter)*** adds to ***FB_list*** this space. Thus the Complete Data Scheduler can use this space to place next results or input data.

The Complete Data Scheduler continues with the next execution of kernel k or with the next kernel execution until all kernels of cluster c have been executed RF times. Then it continues with the next cluster (assigned to the same FB set).

In order to simplify addressing, data and results allocation should be regular. The Complete Data Scheduler stores in the FB data and results of all kernels in the application for RF consecutive iterations. To maintain regularity, data and results are allocated from the addresses where was placed previous iteration of them (Figure 5).

Sometimes a data or result does not fit in any free block, so to improve memory usage the Complete Data Scheduler split it into two or more parts, and as a consequence the access to it is complex.

This algorithm achieves a regular data and result allocation as shown in Figure 5. It tries to have the maximum free space together to avoid having to split data or results into several parts.

6. Experimental results.

In this section we present the experimental results for a group of synthetic and real experiments, in order to demonstrate the quality of the proposed methodology. MPEG is a standard for video compression and ATR stands for Automatic Target Recognition. Synthetic experiments have been generated manually in order to consider. additional features that are not present in the analyzed real applications.

The experiments differ in data dependencies, number of kernels, number of clusters, and data and result sizes as shown in table 1.

The Complete Data Scheduler minimizes memory transfers. We have calculated the improvement in time achieved due to contexts and data reuse.

The Data Scheduler [5] and the Complete Data Scheduler is compared with the Basic Scheduler [3], finding the relative execution improvement on Basic Scheduler as shown in figure 6. Where the first column is the Complete Data Scheduler and the second one is the Data Scheduler.

The data scheduling depends on kernel scheduling, data size and available internal memory. We analyze different kernels schedules for different memory sizes as shown in Table 1.

The Complete Data Scheduler always minimizes time avoiding unnecessary transfers. It reduces transfers for examples that could not be improved by Data Scheduler. It reduces context transfers as shown RF and data transfers from/to the external memory as shown DT (Table 1).

We have tested different kernel schedules for a fixed memory size as shown ATR-SLD. We also have tested a fixed kernel schedule but different memory sizes as shown MPEG and MPEG*, ATR-FI and ATR-FI* or E1 and E1*. A bigger memory allows reusing contexts for an increased number of iterations (RF).

Basic Scheduler cannot execute MPEG if memory size is 1K. Whereas, the Data Scheduler and the Complete Data Scheduler achieve MPEG execution with memory size less than 1K.

In order to demonstrate the quality of the proposed

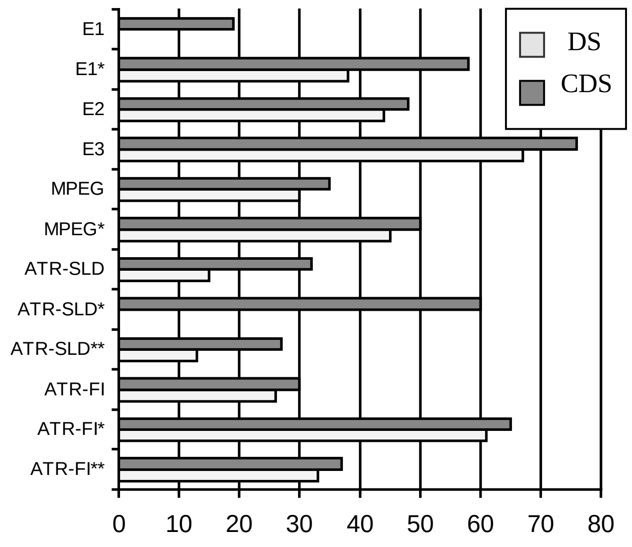

Figure 6. Relative execution improvement (%)

	N	n	DS	DT	RF	FB	DS	CDS
E1	4	4	2.9K	2K	1	1K	0%	19%
E1*	4	4	2.9K	2K	3	2K	38%	58%
E2	4	4	3.3K	0.8K	3	2K	44%	48%
E3	5	5	3K	0.6K	11	3K	67%	76%
MPEG	3	3	3.9K	0.1K	2	2K	30%	45%
MPEG*	3	3	3.9K	0.1K	4	3K	35%	50%
ATR-SLD	3	2	18K	6K	1	8K	15%	32%
ATR-SLD*	3	2	16K	8K	1	8K	0%	60%
ATR-SLD**	4	1	20K	6K	1	8K	13%	27%
ATR-FI	3	2	1.1K	0.3K	2	1K	26%	30%
ATR-FI*	3	2	1.1K	0.3K	5	2K	61%	35%
ATR-FI**	4	1	1.3K	0.3K	2	1K	33%	37%

*(**N**: total number of clusters; **n**: maximum number of kernels per cluster; **DS**: total data size per iteration (input data + intermediate results + final results); **DT**: data transfers avoided per iteration; **RF**: reuse context factor; **FB**: One frame buffer set size; **DS**: Data Scheduler relative execution improvement; **CDS**: Complete Data Scheduler relative execution improvement.)*

Table 1. experimental results

methodology this work is based on the specific allocation algorithm developed in section 5. It achieves that the memory size used is the minimum allowed by the architecture. For all examples no data or result has to be split into several parts. Moreover, it simplifies accesses to FB, as well as, promotes regularity in data allocation.

7. Conclusion.

In this paper we have presented a new technique to improve data scheduling for multi-context reconfigurable architectures. It minimizes data and contexts transfers, reducing the execution time.
The Complete Data Scheduler improves the Data Scheduler results. Data and results reuse is not limited to operations within a cluster. If the FB has sufficient free space, data and results used by other clusters can be retained. It chooses the shared data or results to be kept into FB according to a factor TF, so allowing further reduction transfers from/to the external memory and execution time.
The Complete Data Scheduler maximizes the available free space in the FB. This allows it to increase the number of consecutive iterations (RF), and as a consequence kernel contexts are reused for this number of iterations, so reducing context transfers and minimizing execution time.

The Complete Data Scheduler work is based on a specific data placement policy that tries to simplify accesses to the FB and promote periodicity. Moreover, it achieves that the memory size used is the minimum allowed by the architecture.
The experimental results demonstrate the effectiveness of this technique reducing the execution time compared to the Data and the Basic Scheduler.
Future work will address data management within a kernel, as well as, data and results reuse among clusters assigned to different sets of the FB when the architecture allows it.

8. References

[1] S. Brown, J. Rose. “Architecture of FPGAs and CLPDs: A Tutorial,” IEEE Design and Test of Computer, Vol. 13, No 2, pp. 42-57, 1996.
[2] E. Tau, D. Chen, I. Eslick, J. Brown and A. De Hon, “A First Generation DPGA Implementation”, FDP’95, Canadian Workshop of Field-Programmable Devices, May 29-Jun 1, 1995.
[3] R. Maestre, F. Kurdahi, N. Bagherzadeh, H. Singh, R. Hermida, M. Fernández “Kernel Scheduling in Reconfigurables Architectures”, DATE Proceedings pp 90-96, 1999.

[4] R. Maestre, F. Kurdahi, M. Fernández, R. Hermida, N. Bagherzadeh “A Framework for Scheduling and Context Allocation in Reconfigurable Computing”, Proc International Symposium on System Sybthesis (ISSS’99) pp 134-140 San Jose, California, 1999.

[5] M. Sanchez-Elez, M. Fernández, R. Hermida, R. Maestre, F. Kurdahi, N. Bagherzadeh “A Data Scheduler for Multi-Context Reconfigurable Architectures” Proc international Symposium on System Sinthesys (ISSS’01) Montreal, Canada, October 2001.

[6] H. Singh, M. Lee, G. Lu, F.J. Kurdahi, N. Bagherzadeh and E. Filho, R. Maestre “MorphoSys: Case study of a reconfigurable Computing System Targeting Multimedia Applications”” Proc. Design Automation Conference Proceedings (DAC’00) pp 573-578, Los Angeles, California; May 2000.

[7] R. Maestre, F. Kurdahi, N. Bagherzadeh, M. Fernández, R. Hermida, H. Singh “Analysis and Evaluation of Context Scheduling Heuristic for Multi-Context Reconfigurables Architectures”, 2000 IEEE International Conference on Computer Design. VLSI in Computer & Processors (ICCD’00), Austin, Texas, Sept 2000

[8] M. Kaul, R. Vemuri, S. Govindarajan I. Ouaiss “An Automated Temporal Partitioning and Loop Fission approach for FPGA based reconfigurable synthesis of DSP applications” Proc. 36th ACM/IEEE Design automation conference, 1999, Pages 616 - 622

[9] D. Moolenaar, L. Nachtergaele, F. Catthoor, H. De Man "System-level power exploration for MPEG-2 decoder on embedded cores: a systematic approach", accepted for IEEE Workshop on VLSI Signal Processing Systems, 1997.

[10] Axel Jantsch, Peeter Ellervee, Ahmed Hemani, Johnny Öberg and Hannu Tenhunen “Hardaware/Software Partitioning and Minimizing Memory Interface Traffic”, Procc, Conference on European design automation, 1994, Pages 226 – 231.

[11] P.R. Wilson, M.S. Johnstone, M Neely, and D Boles “Dynamic Storage Application A Survey and Critical Review” IWMM 1995: 1-116