# Kernel Scheduling in Reconfigurable Computing

R. Maestre, F. J. Kurdahi†, N. Bagherzadeh†, H. Singh†, R. Hermida, M. Fernandez

Departamento de Arquitectura de Computadores y Automática
Universidad Complutense - 28040 Madrid, SPAIN
e-mail: maestre@dacya.ucm.es

†Department of Electrical and Computer Engineering
University of California, Irvine, California 92697, USA

## Abstract

*Reconfigurable computing is a flexible way of facing with a single device a wide range of applications with a good level of performance. This area of computing involves different issues and concepts when compared with conventional computing systems. One of these concepts is context loading. The context refers to the coded configuration information to implement a particular circuit behavior. An important problem for reconfigurable computing is the scheduling of a group of kernels (sub-tasks) that constitute a complex application for minimum execution time. In this paper, we show how the different execution orders for these sub-tasks may result in varying levels of performance. We formulate an analytical approach and present a solution for this new problem through this work.*

## 1. Introduction

Reconfigurable computing is an emerging alternative to ASICs and general-purpose processor systems. The main differences between these systems are the range of applicability and the performance for a particular application. ASICs are specifically designed for a particular application, and therefore, they can be highly efficient. On the other hand, general-purpose processors may be used to execute any functionality by programming with software instructions, but their performance is usually lower.

Many DSP and multimedia applications, such as image processing and video compression, demand high performance that sometimes can only be achieved with special-purpose hardware. Reconfigurable computing is a good intermediate solution for these applications. It has a broad range of functionality, enables higher performance than general-purpose processors, and comparable efficiency to ASICs.

New approaches generally introduce new concepts, and therefore, the emergence of reconfigurable computing has presented new problems. One of these is to find an execution order of a set of tasks of an application that minimizes the total execution time. We assume that the timing information of each kernel (subtask within the application) and an execution flow graph of the application are known. Based on these assumptions, we have formulated an algorithm to schedule the kernels so as to optimize performance within hardware constraints.

Previous approaches for scheduling tasks in reconfigurable computing systems are an extension of high-level synthesis techniques, in order to consider specific features of reconfigurable systems [1,2,3,4]. A heuristic based on static-list scheduling, enhanced to consider dynamic area constraints, is proposed in [3], while [4] presents a level based scheduling algorithm. A non-linear programming model approach that also considers synthesis is addressed in [5]. However, these works fail to take into consideration several aspects unique to reconfigurable computing systems, e.g. multiple contexts and data memories, as well as non-constant reconfigurable time.

Although our approach to solve this problem targets one particular reconfigurable system, MorphoSys [6], this approach is quite general in nature and may be easily applied to other reconfigurable systems, e.g. [7,8,9,10]. Several computation-intensive applications such as video compression and target recognition have been successfully implemented with MorphoSys [6], which is a coarse-grain dynamically reconfigurable system.

The paper is organized into five sections. An overview of the MorphoSys architecture is presented in section 2. Next, we define the scheduling problem in the third section. We detail our proposed solution discussed in section 4, which is divided into two subsections: partitioning and scheduling. Finally, we provide some experimental results and conclusions.

## 2. MorphoSys Architecture

MorphoSys architecture is an integrated coarse-grain reconfigurable system. As shown in Figure 1, it consists of an array of reconfigurable cells (RC), a control (RISC) processor, frame buffer (FB), and a DMA controller.

The core of this reconfigurable chip is the RC array, which is composed of 64 reconfigurable cells (RC) arranged as a grid. The architecture of each RC is similar to the data path of a processor. However, there is no control unit, and the control information is instead

This work has been granted by Spanish Government Grant CICYT TIC 96/1071 and by Defense and Advanced Research Project Agency (DARPA) of the Department of Defense under contract F-33615-97-3-c-1043

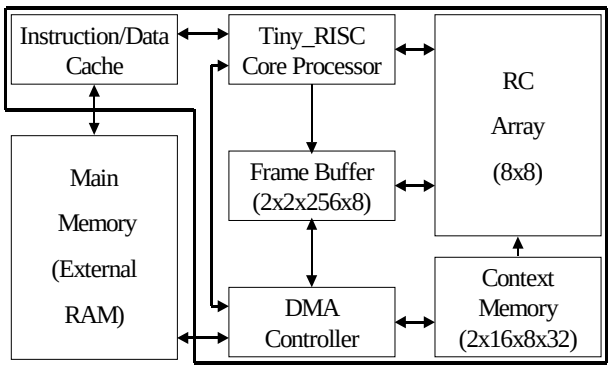


**Figure 1. M1 chip**

included in the context. This context is the binary code that specifies the functionality implemented by the RC array, as well as the active interconnections among the cells. The context word controls the ALU function, the internal multiplexors, the usage of registers, etc. Context information is stored in the context memory (CM). The CM is a two port (1 read, 1 write) memory that can store 32 different 256 bit context words; 16 corresponding to the rows and 16 corresponding to the columns. The particular context to be implemented is loaded from the respective CM location to the context register in the RC. The CM allows dynamic configuration. At the same time that a context is being executed, future configurations can be loaded into the context memory.

The frame buffer is composed of two sets, each having two banks. The RC array may simultaneously access two independent data words stored in different banks of the same set. While the RC array works on data from one set of the frame buffer, the DMA controller enables concurrent data transfer between the other set and the external memory. This way, computation, and data movements (RAM<->FB) can overlap in time if they are carried out over different sets of the frame buffer. The DMA controller also enables data transfer between the external RAM and the context memory.

The MorphoSys system operation is controlled by TinyRISC, a RISC processor based on [11] whose instruction set has been extended with some instructions for specific control of this chip (change of configuration, data movements...).

## 3. Problem Overview and Definition

A typical complex application is composed of a sequence of tasks that are executed repeatedly as a loop. A kernel is a well-defined task of the application that can be independently executed after the previous tasks in the execution flowgraph. Its configuration size must not exceed the size of the context memory. Also, two kernels can not be executed simultaneously. An application example is MPEG, a video compression/decompression standard, whose sequence of kernels for compression is presented in Figure 2a. The kernels involved are motion estimation (ME), motion compensation (MC), discrete cosine transform(DCT), quantization (Q), inverse quantization

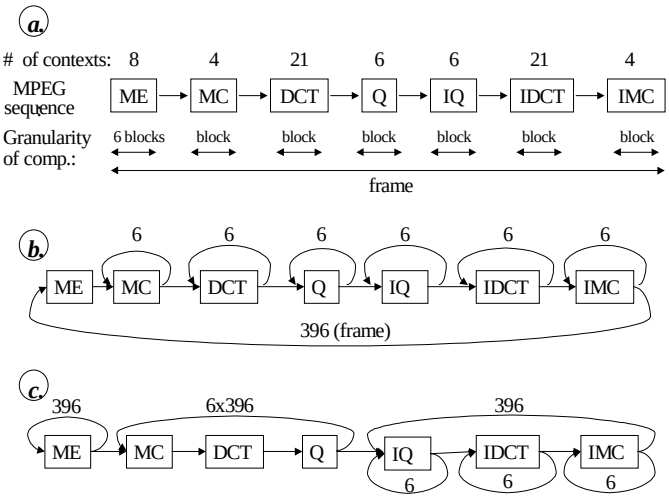


**Figure 2. a) MPEG sequence and granularity, b) a possible schedule of an image frame, c) an alternative schedule**

(IQ), inverse discrete cosine transform (IDCT), and inverse motion compensation (IMC).

Another important concept is the granularity of computation. This refers to the amount of data that is processed by a kernel in a single execution.

Let us consider the MPEG example (Figure 2). All the kernels except ME operate over a block of 8x8 bytes. The granularity of computation for ME is 6 times greater than that of any other kernel. Therefore, it will be necessary to execute all the other kernels six times per each execution of ME in order to process the same size of data. On the other hand, the unit of computational data for the whole sequence is a frame (396 iterations of the whole sequence). This implies that a current frame must be completely processed before starting with the following one.

Although each application imposes some constraints to the execution order (DFG), it is possible to have different execution sequences that result in different total execution times. In order to understand the related aspects, we consider the two cases in Figure 3. If each kernel is executed only once before the execution of the next kernel (case (a)), the context for each kernel has to be loaded as many times as the total number of iterations (time wasted). If some data is re-used by different kernels, it need not be reloaded (time saved). For the other situation (case (b)), each kernel is executed as many times as the total number of iterations before executing the next kernel. For this case, the context for each kernel is loaded only once (time saved). However, as the size of the data produced and used by all the kernels may often exceed the frame buffer capacity, no data reuse is possible (time wasted).

Between these two extreme cases, there may be several other valid solutions. It is possible to execute a kernel a number of times before the data produced exceeds the frame buffer size. This way the data can be reused to some extent and the context is loaded an intermediate

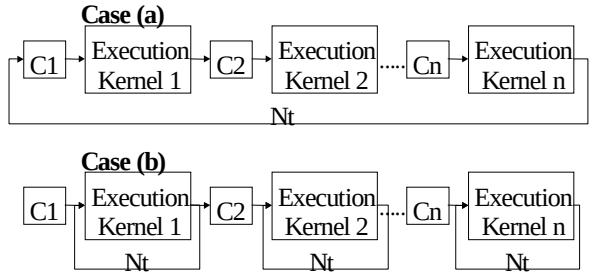


**Figure 3. Two extreme cases of execution sequence for a generic application**

number of times. Moreover, a partition of the execution graph may also improve the final result. Another factor that has to be considered is the overlapping of computation with data transfer.

From the above discussion, it is clear that the major criteria are:

- Context reloading (minimize).
- Data reuse (maximize).
- Computation and data movements overlapping (maximize).

Generally, it is not possible to know in advance which of the feasible solutions is the best. It is necessary to explore the design space, because the above three criteria are mutually conflicting.

### *Problem definition: inputs-outputs*

In order to obtain the best schedule, it is necessary to have the following input information:

- A kernel library which characterizes each kernel for: computation time, input data size, output data size, size of context, granularity of computation, and amount of computation time that may overlap with context loading.
- Reconfigurable system (MorphoSys for the present study) parameters: frame buffer size (4x256x32 bits), context memory size (2x16x256 bits), memory access times, context loading time, data bus width, etc.
- Data-flow graph (Figure 4) for application under consideration, which has information about data dependencies (that constrain kernel execution order).

The solution to the problem is a scheduled linear sequence of kernels with the minimal execution time and its value.

The scheduling algorithm has to take into consideration the following constraints:

- Loading of input data: before execution.
- Loading of context: before execution.
- Write-back /re-use of results: after execution.
- Frame buffer size
- Size of the context memory.
- Overlapping of computation and data movements is possible only if they use different sets of the frame buffer.
- When computation is across rows (columns) context, then a new column (row) context may be loaded simultaneously.

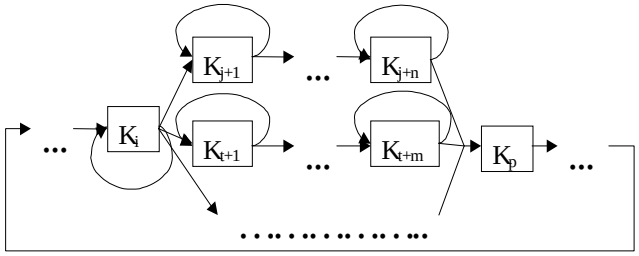


**Figure 4. General data flow graph**

## 4. Proposed Solution

In order to obtain the total execution time for a given application, it is necessary to schedule the kernels. Further, in any application it is possible to find some subsets of kernels that may be scheduled independently of other kernels. We use the term partition to refer to one of these subsets. For example, in Figure 4, the subset $\{K_i,K_{j+1},K_{t+1}\}$ may be executed independently of the rest of the kernels, and therefore, it is a partition, while $\{K_{t+1},K_p\}$ is not, because $K_p$ needs the results produced by $K_{j+1}$, $K_{t+m}$...

It is important to be able to find a good partition. This is because partitioning has a significant influence on the achievable performance, since crucial aspects such as amount of data reuse, the number of context words to be loaded, and overlapping of computation with data transfer are heavily dependent on partitioning. Once a partition has been created, it is possible to schedule it in detail. Consequently, we propose to solve this problem by dividing it into two tasks:

1. Partitioning of the application DFG
2. Scheduling within a given partition

We use a backtracking technique in order to support the search process in both tasks. This process is guided by some heuristic technique that tries to "explore best candidate solutions first". We employ bounding heuristics for an early pruning of the search space. These techniques are explained in sections 4.1 and 4.2.

### *Execution model*

We assume an execution model that satisfies the following constraints (because of architectural issues):

1. Computation can overlap with:
- Context loading if reload of context is in a different part (row/column block) of context memory than what the computation is using.
- Data transfer (RAM-FB) if it concerns data in a different set of the FB.

2. Context loading and data transfer (RAM-FB) can never overlap.

A typical application can be represented as a loop of "*n*" iterations of a sequence of kernels. The first and the last iterations take different time to be executed when compared with the iterations inside the loop body. As "*n*" is always a big number (396 for MPEG), a very small (sometimes zero) error is induced if we only consider the execution of iteration "*i*" in relation with

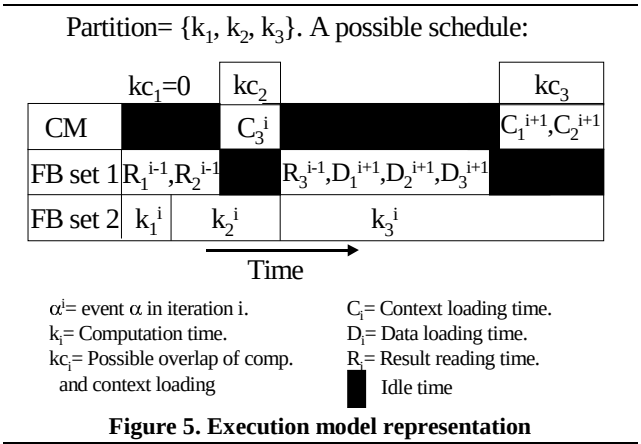


**Figure 5. Execution model representation**

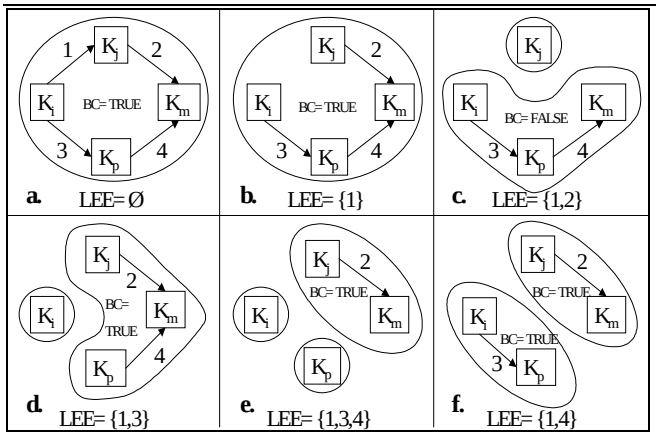


**Figure 6. Some steps of an exploration sequence**

the following "*i+1*", and the previous "*i-1*".
Based on the above considerations, the execution model can be represented as in Figure 5. "*i*" refers to the current iteration that uses data from set 2 of the FB. At the same time, data (results from previous iteration "*i-1*" and input data for next iteration "*i+1*") are transferred between external RAM and set 1 of the FB. Of course, in the following iteration, computation will be carried out on the other set of the frame buffer. The figure also illustrates the fact that context loading can not overlap with any data transfer operation, but may overlap with computation ($kc_i$) if possible. Otherwise, extra time is needed to load contexts.

*Exploration algorithm*

We have formulated an algorithm that explores all the covers of the sets of kernels in a well-defined and orderly manner. To improve the algorithm performance we use the bounding heuristic, which prunes the search space. This algorithm is used for both partitioning and scheduling. The exploration order is governed by the amount of data reuse among the kernels. The lower the data reuse, the higher is the exploration priority. This policy is justified in the following subsections.
The initial step of the algorithm consists of numbering each edge of the DFG in ascending order according to the amount of data reuse between the kernels connected by that edge (Figure 6.a.). If data reuse is the same for several edges, any order is valid, but numbers are not repeated.
In the next step, the edge with the lowest number is erased and added to a list of erased edges (LEE). This results in formation of groups of kernels that have no joining edges (Figure 6.c.-f.). This process is repeated for groups that meet the bounding check (which, in turn, depends on the task).
Each separated group of kernels forms a potential partition whose feasibility has to be checked (a group of kernels is a partition only if it can be scheduled independently of the other kernels). For example, in Figure 6.c. the generated subset $\{K_i,K_p,K_m\}$ is not a partition, since $K_m$ needs the results from $K_j$. Every time a new partition appears, a different cover of the DFG is built, and therefore a different solution is built, so a different solution is explored.

The above mechanism ensures that all possible partitions will be visited, while the bounding check avoids the exploration of solutions that do not surely improve the results.

## 4.1. Partitioning

Partitioning plays an important role in finding the best solution of the search space. The results of scheduling depend on the partition selected. Also, at the time of exploring different partitions, the value for final execution time can only be an estimate.
If we have a means to quickly compute an estimate for execution time, the search space can be explored within a reasonable time. The estimation method has to be independent of the detailed schedule (since it is computed only in the next step). This estimation has to take into account the three criteria introduced in section 3, since the overall problem of finding an execution sequence depends on them.
We assume that computation and data movements (RAM-FB or RAM-CM) can always overlap (this still satisfies the constraints). This way, a lower bound is obtained:

Let $P = \{K_1,...,K_n\}$ be a partition.

$$LB(P) = \begin{cases} \text{if } \sum_{i=1}^{n} kc_i \geq t(C_T) \Rightarrow \\ \Rightarrow MAX\left[\sum_{i=1}^{n} k_i, \sum_{i=1}^{n}\left(D_{1,...,i} + R_i\right) + \frac{t(C_T)}{x}\right] \\ \text{if } \sum_{i=1}^{n} kc_i \leq t(C_T) \Rightarrow \\ \Rightarrow MAX\left[\sum_{i=1}^{n}\left(k_i - \frac{kc_i}{x}\right), \sum_{i=1}^{n}\left(D_{1,...,i} + R_i\right)\right] + \frac{t(C_T)}{x} \end{cases}$$

$C_T$ is the minimum number of context memory words that need to be loaded per loop iteration; $t(C_T)$ is the

time spent to load $C_T$; $x$ is the number of kernel executions between changes of context (in Figure 3, $x=1$ in case (a) and $x=N_t$ in case (b)) ; $k_i$ is the execution time of kernel $K_i$; $kc_i$ (Figure 5) is the portion of computation time that can overlap with context loading ("$t(C_T)-\Sigma kc_i$" is the non-overlapping context loading time); $D_{1,...,i}$ is the time to load the input data from the RAM to the FB if kernel $K_i$ can reuse the data of the previous kernels $K_1,..., K_{i-1}$; $R_i$ is the time to store the results from the FB into the RAM.

When the context size of all the kernels within a partition are known, $C_T$ can be computed easily.

As there are two context memory parts (for rows or columns), *LB* has to include the information regarding both of them: $kc_i$ -> ($kc_{i,R}$, $kc_{i,C}$), and $t(C_T)$ -> ($t(C_{T,R})$, $t(C_{T,C})$). The expression for *LB* is conceptually equal, but there are four possibilities instead of two.

If only one criteria is considered the following partition sizes will produce the best results:

- Context loading: small partitions.
- Data reuse: large partitions.
- Overlap of computation and data transfers: large partitions have better chances of overlap.

Our partitioning process starts with the largest partition as the initial solution, which optimizes the data reuse and computation overlap. Then the exploration mechanism is guided by data reuse, so as to minimize the data loading. This procedure allows good solutions to be found even if the search process is time limited. If the context loading for a partition does not completely overlap with computation, a sub-partition could reduce the total execution time, since $C_T$ is decreased (unless $C_T=0$). However, the data movement time might be increased and the computation overlap could be reduced. Therefore, the sub-partition must be evaluated. If $C_T=0$ no sub-partition will reduce the lower bound, because $C_T$ can not be reduced, and the other two criteria will never improve the result.

Imagine a partition whose data transfer and context loading completely overlap with computation time ($LB(P) = \sum k_i$). Any sub-partition can not reduce the lower bound for the total execution time (see expression for $LB(P)$).

Consequently, if "$LB(P) = \sum k_i$" or "$C_T = 0$" no sub-partitioning improves the lower bound. These checks are jointly expressed in (1):

$$\text{Let } P = \{K_1,..., K_n\} \text{ be a partition and } \{P_1,..., P_m\} \text{ a set of sub-partitions such } P = P_1 \cup ... \cup P_m,, \; P_i \cap P_j = \varnothing, \forall i \neq j. \quad (1)$$

$$\text{If } LB(P) = MAX\left[\sum_{i=1}^{n} k_i, \sum_{i=1}^{n}\left(D_{1,...,i} + R_i\right)\right] \Rightarrow \Rightarrow \sum_{i=1}^{m} LB(P_i) \geq LB(P)$$

Now, suppose that the size of the data used by a given partition is smaller than the size of one set of the FB. Then, the best schedule is obtained as in Figure 5. All the computations are carried out over one set of the FB and all data movements over the other. In this case, computation and data overlap is maximum and the total execution time for this partition equals *LB*. On the contrary, if the total data exceeds the size of one set of the FB, this kind of schedule is not possible and the scheduling has to be done.

Therefore, if a partition, *P*, meets condition (1) and its data fits into one set of the FB, no partitioning of *P* will improve the total execution time. These two conditions form the bounding check for partitioning:

$$\text{Let } P = \{K_1,..., K_n\} \text{ be a partition :}$$
$$\text{If } LB(P) = MAX\left[\sum_{i=1}^{n} k_i, \sum_{i=1}^{n}\left(D_{1,...,i} + R_i\right)\right]$$
$$\text{and } SD(P) \leq (\text{Size of one set of the FB}) \Rightarrow$$
$$\Rightarrow \text{Bounding check} = \text{false}$$
$$\text{Otherwise Bounding check} = \text{true}$$

where *SD(P)* stands for the size of the data used by partition *P*.

The size of the data can be obtained by addition of all the input data and the results generated by all the kernels. However, if a kernel has already been executed, the data that is not used by the following kernels can be replaced by the generated results. Therefore, if "*x=1*" (x is defined in section 4.1.) *SD(P)* can be expressed as:

$$\text{Let } P = \{K_1,..., K_n\} \text{ be a partition.}$$
$$SD(P) = \underset{i\in\{1,...,n\}}{MAX}\left[\sum_{j=i}^{n} d_{i...j} + \sum_{j=1}^{i} r_j\right]$$

*$d_{i...j}$ = Size of input data for kernel $K_j$ except those shared with kernels $\{K_i, ...,K_{j-1}\}$.*
*$r_j$ = Size of the results of kernel $K_j$.*

If "*x>1*" a general expression can be derived from the previous one:

$$SD(P) = \underset{i\in\{1,...,n\}}{MAX}\left[\sum_{j=i+1}^{n} x\cdot d_{i...j} + \sum_{j=1}^{i-1} x\cdot r_j + \underset{l\in\{1,...,x\}}{MAX}\left[(x-l+1)\cdot d_i + l\cdot r_i\right]\right]$$

In this expression, the data used is multiplied by *x* for all the kernels except one of them, say $K_i$. In order to explain this, suppose that $K_i$ is going to be executed *x* times. The first execution of $K_i$ requires "$x\cdot d_i + r_i$". The second one requires "$(x-1)\cdot d_i + 2\cdot r_i$", and so on. The

maximum value thus obtained is the real size of the data used by $K_i$:

$$\underset{l\in\{1,\ldots,x\}}{MAX}\left[(x-l+1)\cdot d_i + l\cdot r_i\right]$$

An important consideration is that within a partition, *P*, reordering of kernels (when possible) can change *SD(P)*. In order to minimize *SD(P)*, the kernel with the largest amount of data used is executed first. This way, the data that have been already used can be replaced by the generated results.

Every time a new partition, *P*, is obtained, *LB(P)* is computed and the lower bounds for all partitions in a complete cover of the DFG are added to obtain a lower bound for the entire DFG with this specific partitioning.

In order to check if this cover for the DFG has the best partitioning solution, its overall lower bound can be compared with a lower bound for the whole search space (SS), *LB(SS)*. The *LB(SS)* considers that the three optimization criteria are met:

$$LB(SS) = MAX\left[\sum_{\forall \text{ kernel}} k_i\,,\ \sum_{\forall \text{ kernel}}\left(D_{1,\ldots,i} + R_i\right)\right]$$

If $\sum LB(P_i)$ for a complete cover and *LB(SS)* are equal, the best candidate has been found, otherwise this comparison provides a means to evaluate the quality of the solution.

## 4.2. Scheduling within a partition

The process of scheduling within a partition can be viewed as an assignment of computations to sets of the FB. For example in Figure 7, computation of kernels 1 and 2 are assigned to set 2 and computation of kernel 3 is assigned to set 1.

We use the term cluster to designate a group of kernels that are assigned to the same set of the FB. In Figure 7 there are 2 clusters: $\{K_1, K_2\}$ and $\{K_3\}$.

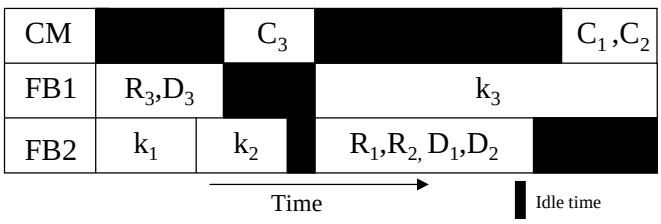


**Figure 7. Scheduling of a partition $\{K_1, K_2, K_3\}$**

The size of the data used by every cluster has to be smaller than one set of the FB and only these clusters will be considered for detailed scheduling.

However in general, there is an important difference between the method to compute the data size *"SD(Cl)"*, for the cluster *Cl*, and the method to obtain the total execution time. This is illustrated in Figure 7. Here, $R_3$ and $D_3$ are loaded while $K_1$, $K_2$ are being executed. So, every cluster execution overlaps with the movement of the results of the previous cluster and the input data of the next cluster. This overlap has to be considered when computing the total execution time.

The minimum number of clusters for a partition *P* is "*SD(P)/(size of FB set)*", and the maximum is the total number of kernels.

The schedule is determined in a similar way to the partitioning. The largest cluster is considered first and the exploration algorithm is applied to it. The largest cluster maximizes the computation and data overlap, but if its size is too big, it has to be partitioned, with consideration for data reuse.

Notice that context loading has already being taken into account during partitioning and any variation of clustering, or even a different execution order, will not modify the context loading time.

The execution time, *ET*, for a given partition, *P*, is:

$$P = \{Cl_1, Cl_2, \ldots, Cl_{NC}\};\ Cl_i = \{K_{i,1}, K_{i,2}, \ldots, K_{i,n(i)}\}$$

$$ET(P) = \frac{t(C_T)}{x} + \sum_{i=1}^{NC} MAX\left[\sum_{j=1}^{n(i)} kn_{i,j}\ ,\ \sum_{j=1}^{n(i-1)} R_{i-1,j} + \sum_{j=1}^{n(i+1)} D_{i+1,(1,\ldots,j)}\right]$$

where $kn_{i,j}$ is the portion of computation time that does not overlap with context loading; $\alpha_{i,j}$ is the variable, $\alpha$, corresponding to the kernel $K_{i,j}$; the other symbols have the usual meaning.

For scheduling within a partition, the bounding check is the same as the stopping criteria. If the execution time for a given partition *P*, *ET(P)*, equals its lower bound, *LB(P)*, there is no other clustering that can improve the result.

Moreover, if a partition, $P_i$, is scheduled and "$ET(P_i) \leq LB(P_j)$", $\forall$j, then the best scheduling has been found.

## 5. Experimental results

In this section, MPEG (a practical application) is used to demonstrate the quality of the proposed methodology. Some of the best results generated during the search are presented in Table 1. One solution that was not explored (because of search space pruning through bounding check) is also included in the table to illustrate the validity of the bounding check.

Although the best solution is obtained quite fast (in the second iteration), the partitioning process can not be terminated early, because *LB(SS)* is never reached.

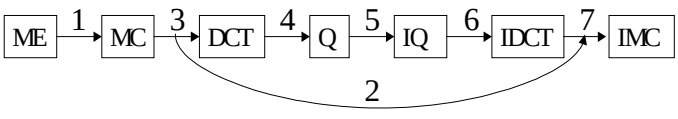


**Figure 8. Ordering of edges for MPEG.**

| Exploration algorithm iteration | Cover | LEE | $\sum LB(P_i)$ *(clock cycles)* | Execution time (after scheduling) |
|---|---|---|---|---|
| 1 | {ME,MC,...,IMC} | Ø | 5110 cc | *NNS* |
| 2 | {ME} {MC,...,IMC} | {1} | 4941 cc | 4941 |
| 15 | {ME} {MC,DCT,Q} {IQ,IDCT,IMC} | {1,2,5} | 5080 cc | *NNS* |
| 30 | {ME,...,Q} {IQ,...,IMC} | {2,5} | 5036 cc | *NNS* |
| Not explored | {ME} {MC} {DCT} {Q} {IQ} {IDCT} {IMC} | {1,2,3,4,5,6,7} | 5806 cc | |

*LB(SS)= 4894 cc.; NNS= not necessary to schedule.*

**Table 1. Experimental data for MPEG**

Moreover, we found that the bounding check reduced the search space from 64 different covers to only 31. For example, the solution with *LEE={1,2,3,4,5,6,7}* (case (b) in Figure 3) is not explored.

Additionally, for the second partitioning solution, the execution time (obtained after detailed scheduling) is lower than the lower bound for all other partitioning solutions. Therefore, we do not need to perform detailed scheduling for any other cover, since it is guaranteed that no other solution can have lesser execution time.

## 6. Conclusions

In this paper, we have presented a solution to the problem of scheduling of application kernels, for the MorphoSys reconfigurable system. The various aspects related to this problem have been discussed, and a methodology to solve it has been proposed. The optimal solution is always found through the exploration of the pruned search space. On the other hand, if the process times out, it is likely that a good solution has already been found (since better candidates are evaluated earlier). The exploration algorithm is quite straightforward and the equations can be quickly computed, facilitating its implementation in software.

## References


[1] I.Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul and R. Vemuri, "An Integrated Partitioning and Synthesis system for Dynamically Reconfigurable Multi-FPGA Architectures", 5th Reconfigurable Architectures Workshop, 1998 (RAW'98)

[2] M. Vasilko and D. Ait-Boudaoud, "Architectural Synthesis Techniques for Dynamically Reconfigurable Logic", 6th International Workshop on Field-Programmable Logic and Applications, FPL '96 Proceedings, p.290-296

[3] M. Vasilko and D. Ait-Boudaoud, "Scheduling for Dynamically Reconfigurable FPGAs", in Proceeding of International Workshop on Logic and Architecture Synthesis, IFIP TC10 WG10.5, Grenoble, France, Dec. 18-19, 1995, pp. 328-336

[4] K. M. GajjalaPurna, D. Bhatia, "Temporal partitioning and scheduling for reconfigurable computing", Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines, 1998, pp. .329-330.

[5] M. Kaul and R. Vemuri, "Optimal Temporal Partitioning and Synthesis for Reconfigurable Architectures", Proceedings of the DATE, p. 389-396, 1998.

[6] H. Singh, M. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, T. Lang, R. Heaton and E. M. C. Filho, "MorphoSys: An Integrated Re-configurable Architecture", Proceedings of the NATO Symposium on System Concepts and Integration, Monterey, CA, April 1998

[7] E. Waingold et al, "Baring it all to Software: The Raw Machine", IEEE Computer, p. 86-93, Sep 1997

[8] E. Tau, D. Chen, I. Eslick, J. Brown and A. Dehon, "A First Generation DPGA Implementation", Third Canadian Workshop of Field-Programmable Devices, May 29 – Jun 1, 1995

[9] E. Mirsky and A. Dehon, "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources", Proc, of IEEE Symposium on FPGAs for Custom Computing Machines, IEEE CS Press, p. 157-166, 1996

[10] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Re-configurable Co-processor", Proc. of the IEEE Symposium on FPGAs for Custom Computing Machines, April 1997

[11] A. Abnous, C. Christensen, J. Gray, J. Lenell, A. Naylor and N. Bagherzadeh, "Design and Implementation of the Tiny RISC Microprocessor", Microprocessor and Microsystems, Vol. 16, No, 4, p. 187-194, 1992