A Framework for Scheduling and Context Allocation in Reconfigurable Computing

R. Maestre, M. Fernandez, R. Hermida, N. Bagherzadeh[†]

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 emerging as a viable design alternative to implement a wide range of computationally intensive applications. The scheduling problem becomes a really critical issue in achieving the high performance that these kind of applications demands. This paper describes the different aspects regarding the scheduling problem in a reconfigurable architecture. We also propose a general strategy in order to perform at compilation time a scheduling that includes all possible optimizations regarding context (configuration) and data transfers. In particular, we focus especially on the methodology and mechanisms to solve the context scheduling. Some experimental results are presented to validate our assumptions. Finally, the problem of data transfers is only formulated and will be addressed in future work.

1. Introduction

Reconfigurable computing is consolidating itself as a real alternative to ASICs and general purpose processors. The main advantages of reconfigurable computing derive from its unique combination of broad applicability (like general-purpose systems), and achievable performance (comparable to special purpose circuitry). Recently, this area of computing has reached performance figures that enable it to appear as a serious competitor in a DSP and multimedia applications market. However, the scheduling of the different tasks must be carefully analyzed to efficiently exploit all the capabilities of a reconfigurable system. It is especially crucial in real time applications.

The scheduling problem in reconfigurable computing is relatively new, and has only been addressed in a few investigations. Most of them are versions of existing high-level synthesis techniques extended in order to consider specific features of reconfigurable systems, such as the reconfiguration time [1,2,3,4]. A heuristic technique based on static-list scheduling, enhanced to consider dynamic area constraints, is proposed in [3], while [4] presents a level-based scheduling algorithm. A new approach to the problem is presented in [5,6], where an ILP model is applied to the temporal partitioning of a task graph. However, none of the existing techniques considers key architectural features such as multiple levels of reconfiguration and multiple data caches.

In [7] we presented a new approach to scheduling in reconfigurable computing. Given a task graph showing data dependencies, together with some additional information (task execution time, data sizes, …), the goal of that work was to find the task schedule having the optimal execution time. To do a realistic investigation, our work targeted a particular coarse-grain reconfigurable device, MorphoSys [8], that imposes a set of architectural constraints, and provides dynamic reconfiguration. Nevertheless, the proposed approach is quite general in nature, and can be adapted to other reconfigurable systems, provided that their key architectural characteristics are known.

While the above mentioned research focused on finding the best task execution order, other very important aspects, such as the scheduling of data transfers and reconfiguration (context) updates were only optimistically estimated. However, even if a highly efficient task schedule is found, the impact of reconfiguration and data scheduling should not be neglected. If a large part of all these context/data transfers cannot be performed in parallel with effective computation, as assumed during the task scheduling, the expected performance may severely decrease. So, in this work, besides summarizing the task scheduling methodology, we present a framework where all the possible optimizations regarding configuration and data transfers are considered. Such optimizations are done at compilation time, so that runtime overhead is avoided. We especially focus on context scheduling in order to maximize the context loading that overlap with computation. The efficient management of the available context storage, from the point of view of allocation, replacement, and fragmentation, is also considered. The data management is only formulated and will be addressed in future work.

This paper is organized into 6 sections. First, we provide an overview of the MorphoSys architecture, as well as the framework within which our work is being developed. Next, we present the kernel scheduler. In section 4, the details of the context and data scheduler are discussed. Moreover, we present the mechanism to solve the context scheduling problem. Finally, we provide the experimental results and conclusions.

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

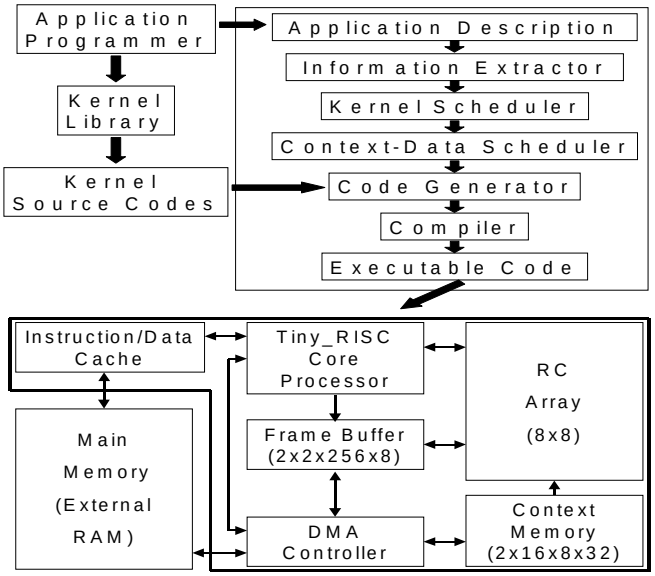

Figure 1. Framework and M1 chip.

2. Overall framework

A typical complex multimedia application is composed of a sequence of macro-tasks that are repeatedly executed within a loop. We use the term *kernel* [7] to refer to a well-defined coarse-grain task of the application that can be independently executed after the previous tasks in the execution flow graph. Its configuration information must not exceed the size of the context storage. Also two kernels can not be executed simultaneously. Discrete Cosine Transform (DCT) may exemplify one of these typical tasks. While some kernels like DCT can be used in many different applications, other ones may be application-specific procedures. However, taking these kernels as the logic entities to be scheduled is usually a good choice, because a further division of them implies a lot of data transfers between parts.

An overview of the proposed framework is shown in Figure 1. The application description is given in C code. This code is written in terms of kernels that are available in a kernel library. The kernel programming is equivalent to specifying the mapping of computations to the target architecture, and it is done only once. After a sequence of transformations and optimizations, which encompass most of this research and will be commented on later, the generated code could be executed by the M1 processor. M1 is the first implementation of MorphoSys, a coarse grain reconfigurable architecture, mainly composed of an array of reconfigurable cells (RC), a control RISC processor, a frame buffer (FB), a context memory (CM) and a DMA controller. The core of M1 is the RC array, which has 64 cells arranged as a grid. Each cell is basically a processor whose control information is provided by an internal configuration register. Such registers are loaded from the CM, which can store 32 context words (of 256 bits each): 16 words corresponding to rows of the RC array, and 16 to columns. When a row (column) word of the CM is copied to a cell, all the cells of the same row (column) will share the same context. The FB has a dual structure (it is composed of two *sets*) to allow simultaneous execution and

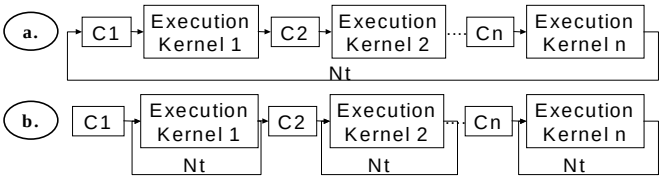

Figure 2. Two extreme cases of execution sequence for a generic application.

data transfer. While the RC array is working on data from one set of the FB, the DMA enables data transfer between the other set and the external memory. The main architectural issues of MorphoSys that are relevant for scheduling are summarized as follows:

(a) Computation on the RC array can overlap with:

– Context loading on CM row (column) positions, while computation is accessing column (row) positions,

– Data transfers (RAM ↔ FB) to/from one set of the FB, if computation is using the other set.

(b) Context loading and data transfer can never overlap.

A detailed description of MorphoSys is given in [8].

Taking into account the constraints discussed above, we can now address the processing of the application code. The first step is information extraction. This means to generate the information needed by the kernel scheduler, including kernel execution times, data reuse among kernels, as well as the data size and context size of each kernel.

The kernel scheduler explores the design space to find a sequence of kernels that minimizes the execution time. Although each application imposes some constraints on the execution order (data/control flow), there may be a lot of valid kernel sequences with different execution times. Figure 2 shows two extreme schedules of a hypothetical application. If each kernel is executed only once before the execution of the next one (case *a*), each kernel's context has to be loaded as many times as the total number of iterations, N_t. Conversely, if some data are reused by several kernels, data reloading time can be saved. In the other extreme, if each kernel is executed N_t times before executing the next one (case *b*), each kernel's context has to be loaded only once. However, as the size of the data produced and used by all the kernels is likely to exceed the FB size, data reuse may be low. Between these two extreme alternatives, the kernel scheduler can explore many intermediate solutions.

The notion of a *partition* plays a key role in kernel scheduling. We understand a partition as a set of kernels that can be scheduled independently of the rest. Finding good partitions eases the problem, because scheduling within a partition only considers internal kernels and data dependencies. In other words, a partition is scheduled as an independent application. Thus, data reuse as well as overlapped computation and data transfer are possible only within a partition. Figure 3a uses a time diagram to show a simple schedule of a partition composed of three kernels $\{K_1, K_2, K_3\}$. The amount of time available to interchange information among the RAM, the FB, and the CM as well as the time intervals where context loading can overlap with execution, are also shown. Figure 3a shows only the scheduling of kernels in iteration "*i*". For the iteration "*i+1*" the roles of both FB sets will be interchanged.

The context and data events are planned by the context/data scheduler (Figure 3b). Overlapping of execution and information transfer is the key aspect to be optimized. However, it is also important to consider several conditions that help to preserve code simplicity, and limit

the overhead introduced by scheduling.

In the code generation step, the final version of the application is created. It includes the code of all the kernels, together with the scheduling specification. Finally, the compiler generates the executable code.

3. Kernel scheduler

This subject is the focus of the work proposed in [7]. However, for the sake of completeness and clarity a brief summary is presented here. From the point of view of performance, the quality of a partition P can be estimated without scheduling. Assuming that overlapped computation and data transfer is always possible, the computation time can be directly obtained through the evaluation of an expression. Furthermore, this expression gives a lower bound for the actual execution time, *LB(P)*.

Let $P = \{K_1,..., K_n\}$ be a partition. Then $LB(P) =$

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

where C_T is the minimum number of context memory words that needs to be loaded per iteration loop; $t(C_T)$ is the time spent in loading C_T; x is the number of kernel executions between changes of context; k_i is the execution time of kernel K_i; kc_i is the portion of computation time that can overlap with context loading (it limits the number of context words that can be loaded); $D_{1,...,i}$ 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 required to store the results from the FB into the RAM. The actual execution time equals the lower bound in the case of a big enough frame buffer. Otherwise, it is necessary to optimize the scheduling. In order to simplify the search process we divide the problem into two tasks: i) partitioning of the task graph, and ii) scheduling within a partition. The first task explores the design space in search of the best candidate partition. This is either the partition, P, with the lowest *LB(P)*, when the whole space has been explored, or the partition whose lower bound equals the lower bound for the whole search space, *LB(SS)*:

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

LB(SS) is obtained with the assumptions of maximal overlapping, as well as zero context loading.

Once the best candidate partition has been found, it is scheduled. If the execution time is lower than any other partition lower bound, the best solution has been found. Otherwise, it is necessary to schedule the next partition, in ascending order of *LB(P)*, until the previous condition is met. The scheduling can be viewed as an assignment of computations to sets of the frame buffer (Figure 3a). Some groups of kernels are assigned to one set of the frame buffer, while the other set is used for data transfers. We call *cluster* to a group of kernels that are assigned to the same set of the frame buffer during scheduling, and are executed consecutively. Only if the size of the data used by all the clusters is smaller than the size of the frame buffer, the schedule is valid. In this case, the execution time is computed through the expression:

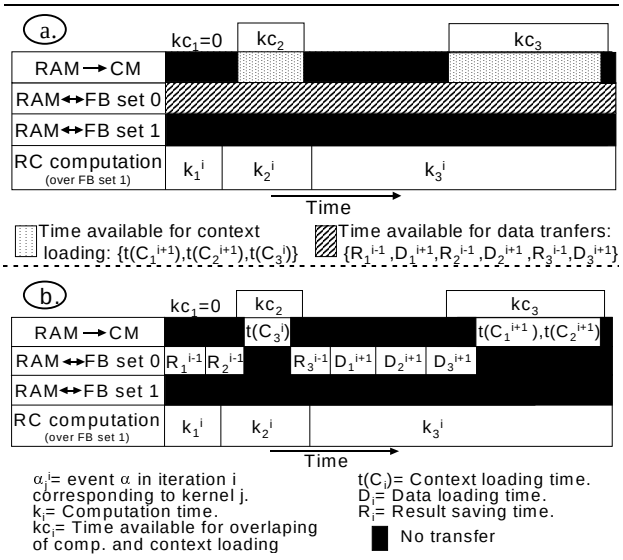

Figure 3. Scheduling within a partition for iteration i: a. Kernel schedule, b. Kernel and Data-context schedule.

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

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

where partition P is composed by NC clusters, and each cluster has a set of kernels; $kn_{i,j}$ is the portion of computation time that does not overlap with context loading; $\alpha_{i,j}$ is the variable α, with its usual meaning, corresponding to the kernel $K_{i,j}$. The schedule with the lowest execution time will be the best solution.

Given the similarity between partitioning and scheduling within a partition (clustering), both are guided by the same exploration algorithm. It is an iterative process whose starting solution is the whole application to be partitioned, or the whole partition to be scheduled, respectively.

4. Context-data scheduler

The kernel scheduler generates a solution (Figure 3a) that only imposes a high-level order (sequence of kernels). However, the context loading and data movements have not been scheduled (compare Figure 3a and 3b). During the kernel scheduling task, it is estimated that these events are optimally scheduled. So, if the scheduling of these actions is not carefully performed, the efforts of the kernel scheduling solution will be in vane. Similarly to the kernel scheduler, the context-data scheduler has to maximize the overlapping of context and data transfers with computation so that the final execution time is minimized.

The selection of the positions for the data in the frame buffer and the context words in the context memory is closely related to the scheduling. Firstly, a bad use of the memory resources may force some avoidable reloading of data to be performed. Secondly, it may have the drawback of generating memory fragmentation, which cause two negative impacts on the execution time: memory access initialization time and the generation of a more complex code (next step in the framework). Every time a loading

process has to be started, a new loading statement is introduced in the code. This overhead may delay the execution of the following instructions and increase the final execution time.

There are more considerations regarding the complexity of the final code. As stated before, the schedule of a typical application can be represented as a loop of a series of kernels. This facilitates the translation into a programming language. Similarly, from the data and context point of view, it is desirable to obtain a periodical schedule in order that the complexity of the final code is minimized. We will say that this schedule is periodical in time.

The above reasoning can also be applied to spatial recurrence. If the data is placed in the same positions (periodical in space) as it was in the previous iteration, the form of the statements will be the same. If not, the increase in code and the resulting processor overhead that the non-periodical solution introduces makes them worse solutions.

To summarize, it is desirable to find a solution such that:

- Overlapping of context and data transfers with computation is maximized.
- Data and context memory fragmentation is minimized.
- Its schedule is periodical in time and space.

The next step is to tackle the problem of how to schedule data and context movements. The methodology that we propose divides this into three tasks:

1. Distribution of overlapping time between context and data movements.
2. Context scheduling.
3. Data scheduling.

The first step is taken so that the two following tasks, may be addressed as separate problems. It distributes, between context and data movements, the time available for overlapping with computation. This decision divides the problem into two sub-problems, and is based on the small interaction between both tasks.

The next two subsections describe the first and second step in this methodology. The case of data scheduling will be addressed in a future work.

4.1. Distribution of overlapping time

There is an important difference between context and data storage. A kernel always uses the same context words, whereas the data may change from one execution to another. If some context words are kept in the context memory until the next iteration execution, their reloading will be avoided. Furthermore, if time for overlapping is available during the execution of a kernel, it can be used to perform some loading of context words of other kernels. Therefore, this loading is skipped so that some time will be saved in case there is no time for overlapping in future executions. We could say that the context gives more "optimization choices" than data transfers, and so it is good to use as much time as is available for context loading. On the contrary, data transfers usually use the same amount of time. Therefore, $TAOC^{R(C)}$ (total computation time available for overlapping with context loading corresponding to rows (columns)) is limited to the time that is not used by data movements, $\sum_{\forall i \mid K_i \in P} \left(k_i - \left(D_i + R_i\right)\right)$.

Furthermore, $TAOC^{R(C)}$ is also limited by the free time of the row (column) CM locations, $\sum_{\forall i \mid K_i \in P} kc_i^{R(C)}$. Finally, the number of free positions in the context memory also limits the number of context words that can be loaded. The time available is $\sum_{\forall i \mid K_i \in P} \left(SCM^{R(C)} - C_i\right)$, where $SCM^{R(C)}$ is the size of the context memory corresponding to rows (columns). All these ideas can be simultaneously expressed in the following equation:

$$TAOC^{R(C)} = MAX\left[0; MIN\left[\sum_{\forall i \mid K_i \in P} k_i - \sum_{\forall i \mid K_i \in P} \left(D_i + R_i\right); \sum_{\forall i \mid K_i \in P} kc_i^{R(C)}; \sum_{\forall i \mid K_i \in P} t\left(SCM^{R(C)} - C_i\right)\right]\right]$$

The maximum is used to ensure that $TAOC^{R(C)}$ is never lower than zero.

In summary, data movements use as much time as they need and the free time (if available) is used by context loading.

The experimental results have shown that, the way *TAOC* is split between rows and columns has no impact on the overall execution time (see Table 2). Because of this fact and for the sake of simplicity, in the next sections we will assume a single type of CM locations without distinguishing between rows or columns. Each of them will need a separate and equivalent process. Hence, we will use *TAOC* to refer to either $TAOC^R$ or $TAOC^C$.

4.2. Context scheduling

As context words are reused in each iteration, it is good to keep as many words as possible in the memory, while performing the minimum number of context reloadings that do not overlap with computation. Therefore, we first have to establish which context words have to be loaded, and which may stay in the memory, so that the total execution time is minimized. At the same time, the decision over which loadings will overlap with computation is made. The process just described will be called *context selection*, and its goal is to minimize the total number of context loads that do not overlap with computation.

Once the specific context words that will be reloaded have been chosen, it is necessary to select the memory locations that these words will occupy. This *context allocation* process takes into account the initialization time that results when a block is placed in non-consecutive locations (memory fragmentation). Context selection has bigger impact on the overall execution time than context allocation, so it will be considered first.

4.2.1. Context selection. A mathematical model has been contrived to state the problem, and this has been solved through two different approaches. The first one explores all feasible solutions in search of an optimal result, and its role is to validate the results reached by the second one, which is a heuristics.

Let's consider the example shown in Figure 4 to illustrate several definitions that will be used later. $P=\{K_1, K_2, K_3\}$ is a scheduled partition whose context loading we want to schedule. The corresponding numbers of context words are $C_1=18$, $C_2=14$, $C_3=10$. In this case we suppose $SCM=32$. As the total number of context words is *42*, it is necessary to reload some of them.

We present our method through a matrix *PREP_EX*. The "*i-th*" column represents the number of context words that are in the CM corresponding to kernel "*i*". Different rows stand for different instants. The first row takes into

$P=\{K_1, K_2, K_3\}$; $C_1=18$, $C_2=14$, $C_3=10$

$PREP\_EX=$

$$\begin{pmatrix} \boxed{18}^{0} & 12 & 2 \\ \textcircled{18}^{0} & 12 & 2 \\ 16 & \boxed{14}^{2} & 2 \\ 14 & \textcircled{14}^{2} & 4 \\ 10 & 12 & \boxed{10}^{0} \\ 10 & 12 & \textcircled{10}^{0} \end{pmatrix} \Rightarrow \begin{matrix} PREP= \begin{pmatrix} \boxed{18} & 12 & 2 \\ 16 & \boxed{14} & 2 \\ 10 & 12 & \boxed{10} \end{pmatrix} \\ EX= \begin{pmatrix} \textcircled{18}^{0} & 12 & 2 \\ 14 & \textcircled{14}^{2} & 4 \\ 10 & 12 & \textcircled{10}^{0} \end{pmatrix} \end{matrix}$$

Figure 4. Context configuration arrays.

```
PREP_computation(i)
{ PREP_{i,i} = C_i; S = C_i; j = i + 1;
while(j ≠ i)
    { PREP_{i,j} = MIN[EX_{i-1,j}; SCM − S];
    S = S + PREP_{i,j};
    j = j + 1; } }
```

Figure 5. Computation procedure for a *PREP* row.

```
EX_computation(i)
{ EX_{i,i} = C_i; ΔEX = S = 0; j = i + 1; δ = TAOC_i;
while(δ > 0 and j ≠ i and S < SCM)
    { EX_{i,j} = MIN[C_j; PREP_{i,j} + δ];
    if(S = S + EX_{i,j}) > SCM
        { EX_{i,j} = EX_{i,j} − (S − SCM);
        S = SCM; }
    ΔEX = ΔEX + EX_{i,j} − PREP_{i,j};
    δ = TAOC_i − ΔEX; j = j + 1; }
while(j ≠ i)
    { EX_{i,j} = MIN[PREP_{i,j}; SCM − S];
    S = S + EX_{i,j};
    j = j + 1; } }
```

Figure 6. Computation procedure for an *EX* row.

account the context words that are in the CM just before the execution of K_1 (the context words have been *prepared* for execution), while the second one represents the context words loaded in the CM just at the end of the kernel *execution*. Similarly, the third and fourth rows, and the fifth and sixth rows have the same role with respect to K_2 and K_3. This matrix is split into *PREP* (odd rows) and *EX* (even rows) which group the rows concerning *preparation* (before execution), and those ones related with *execution* itself. Since during the execution of K_i, its whole context has to be in the CM, the circled number, $EX_{i,i}$, always equals C_i. Similarly to $EX_{i,i}$, $PREP_{i,i}$ (within a rectangle) always equals C_i, as the context words of K_i are "prepared" for its imminent execution. Therefore, $MAX(0\ ;\ PREP_{i+1,j} - EX_{i,j})$ is the number of context words of K_j that have been loaded without any overlap with execution, just after the execution of K_i. On the contrary, $MAX(0\ ;\ EX_{i,j} - PREP_{i,j})$ is the number of context words of K_j that have been loaded with K_i execution overlap. Thus, the goal of context selection is to minimize the number of context words that are reloaded without overlapping with computation, $\sum_{\forall i,j} MAX\left(0; PREP_{i+1,j} - EX_{i,j}\right)$.

a) Mathematical model

We present here a mathematical model whose solutions (*EX* and *PREP* arrays) are valid context distributions. Two cost functions will be used to quantify the quality of the solutions. From the discussion in the previous subsection, it is clear that all the context words have to be in the CM during the whole execution of a kernel. Thus:

$$if\ K_i \in P, then:\ EX_{i,i} = C_i, \forall i;\ \ and\ \ \ PREP_{i,i} = C_i, \forall i$$

The total number of words in the CM can not exceed its size, *SCM*. Moreover, there are no free positions, since they would be used in order to minimize the number of context loads. So:

$$\sum_{\forall j} EX_{i,j} = SCM, \forall i;\ \ and\ \ \sum_{\forall j} PREP_{i,j} = SCM, \forall i$$

The number of context words allocated to a kernel, K_j, will never exceed the total, C_j. So:

$$EX_{i,j} \le C_j, \forall i \ne j;\ \ \ and\ \ \ PREP_{i,j} \le C_j, \forall i \ne j$$

Only a portion of *TAOC* ($TAOC_i$) is assigned to each kernel, K_i. It may be used for context loading of $K_j|j\ne i$, and it cannot be greater than kc_i and $t(SCM^{R(C)}-C_i)$. Moreover, *TAOC* is a limit for $\sum_{\forall i} TAOC_i$:

$$TAOC_i = MIN\left[kc_i\ ;\ t\left(SCM - C_i\right)\right], \forall i$$

$$TAOC \ge \sum_{\forall i} TAOC_i$$

$TAOC_i$ limits the time assigned for overlapping with k_i:

$$TAOC_i \ge \sum_{\forall j \ne i} MAX\left[0\ ;\ EX_{i,j} - PREP_{i,j}\right], \forall i$$

Any proposed solution that meets all these expressions is a feasible solution. In order to evaluate the quality of this solution we will use a cost function given by the number of context loads that do not overlap with computation.

$$CF_1 = \sum_{\forall i} \sum_{\forall j \ne i} MAX\left[0\ ;\ PREP_{i+1,j} - EX_{i,j}\right]$$

If several solutions have the same "CF_1", the best one has the minimum number of context loads that overlap with computation. So, the secondary cost function ("CF_2") is:

$$CF_2 = \sum_{\forall i} \sum_{\forall j \ne i} MAX\left[0\ ;\ EX_{i,j} - PREP_{i,j}\right]$$

b) Heuristic approach

During the complete execution of a kernel, all its context words are in the CM. So, if it is possible to overlap context loading with computation, new context words have to be stored in positions that are not occupied by the current kernel. If a kernel has just been executed, its context words have the highest probability of being replaced, since before its next execution, all the other kernel has to be executed. If there are not enough locations, the previous kernel in the execution flow will also be used. The rest of the context words remain in the CM. This way, we can compute the row, "*i*", of *EX*, from the row, "*i*", of *PREP*:

$$EX_{i,j} = MIN\left[C_j; PREP_{i,j} + \delta\right]$$

where δ is the portion of $TAOC_i$ that has not yet been used. The same reasoning is applied if the context loading does not overlap with computation. However, all the CM locations are then candidates. The first locations replaced belong to the kernel that has just been executed, and the previous ones are used if necessary. Consequently, we can compute the row, "*i*", of *PREP* from the row, "*i-1*", of *EX*:

$$PREP_{i,j} = MIN\left[EX_{i-1,j}; SCM - S\right]$$

where *S* is the number of locations that are currently being used.

Both expressions are used within two procedures, so that one *EX* row and one *PREP* row are generated. They are shown in Figures 5 and 6. Notice that each row is generated from the previous row. Therefore, it is necessary to produce a starting row. The process is repeated until a periodical

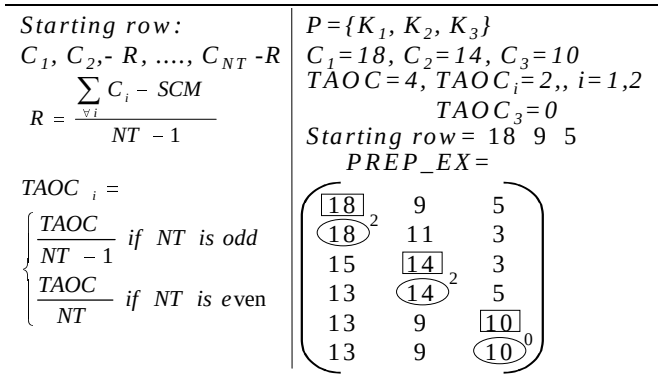

Figure 7. Heuristic example.

solution is found, or otherwise it stops after a number of iterations and selects the best solution found. The starting row is computed by allocating the CM positions, so that the number of loadings for each kernel is the same. We can imagine that there is a free block in the CM, and every time a kernel is going to be executed, its own context words occupy the free block. In the case "*TAOC=0*", this kind of solution will lead to a periodical one. In the general case, *TAOC*≥*0*, the heuristics will build the solution.

Regarding $TAOC_i$, we have found from experiments that the best solution is obtained when *TAOC* is divided as equally as possible among all $TAOC_i$, if the total number of kernels (*NT*) is even. Otherwise (*NT* odd), the division is carried out among *NT-1*, as in Figure 7. *TAOC* is expressed in integers of one context word loading time, and in *PREP_EX*, $TAOC_i$ is represented as a superscript associated with the circled elements.

4.2.2. Context allocation. The context selection task obtains a solution that provides information about the optimal configuration of context words in the memory (periodical "in time"), as well as whether the context loading overlaps with computation or not. The next step is to decide where to place each context word so that memory fragmentation is minimized, while providing a periodical solution "in space".

In the context selection solution there are some context words that remain in the CM (static context words) during all the iterations, and some others that are reloaded (dynamic). We can always choose to allocate the dynamic positions in a single block (see Figure 8). Therefore, the problem is reduced to studying the allocation of the dynamic context words.

The specification of a context selection configuration includes complete information about which context words have to be replaced and which ones have to be placed. This fact constraints too much the context allocation task. For example, in Figure 8 the number of context words of kernel K_2 is decreased in 4 words from the fourth row to the fifth row of the dynamic *PREP_EX*. Hence, these free positions are replaced by four context words of the third kernel K_3. However, a better solution could be that K_3 uses one position of K_1. In order to consider all the possibilities, we will not specify how many context words disappear from one row to the next one. Once a kernel has been executed, all its dynamic positions are free, and the context allocation task decides which to use. We use a representation of all the possibilities as presented in Figure 8 (set of equivalent dynamic *PREP_EX*).

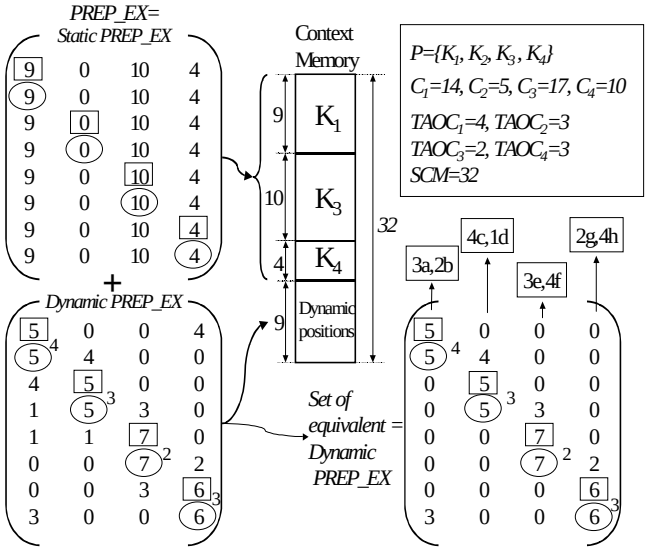

Figure 8. Context allocation example.

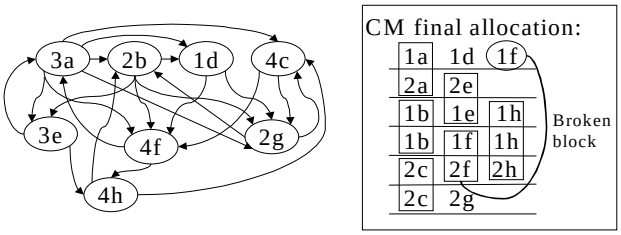

Figure 9. Block dependency graph.

The allocation process now has to avoid the fragmentation of the context blocks that are loaded as a whole in one step. Thus, a graph (Figure 9) representing the dependencies among the context blocks is generated, so that all possibilities can be explored in an orderly manner. A node in the graph represents a dynamic block. An edge is added between the nodes if the source node disappears from the set of dynamic *PREP_EX*, and the destination node appears later.

Let's consider the set of dynamic *PREP_EX* array in Figure 8. If we suppose that the context loadings that are necessary in each row are loaded consecutively, but independently of the other rows, there are 8 context blocks. These blocks are identified with letters, from *"a"* to *"h"*, and the number indicates how many context words the block has. The graph of the context block dependencies is represented in Figure 9. For example, the block *"2g"* is connected to *"2b"* and *"4c"*. However, for *"2g"*, the dependencies can not go further than the first row of *PREP_EX* (i.e. *"3e"*), because all its dynamic positions are occupied. The process is now divided into two steps. First, we look for a periodical solution, because it is the most important condition. Then, we try to place the context blocks in consecutive positions. In order to find a periodical solution, the graph is explored from the sources until it reaches a context word that belongs to the starting kernel (i.e. *{a,e,a}*). If this is not possible, we perform backtracking. The context words that are added to a path are removed from the graph. If a non-periodical path is built, we perform backtracking to look for a periodical path. The whole process is applied until there are no context words in the graph. Afterwards, we try to place the paths in

Experimental data	$TAOC^{R(C)}$	Context loads **Optimal**	Context loads (error) / Fragmentations / Nº of iteration found **Heuristics**		Context loads **Random generation**
			$TAOC_i$ distribution: Uniform	Non-uniform	
NK=3;ΣC_i=49	12	20	20 (0%)/ 0 / 1	20 (0%) / 0 / 1	29
NK=4;ΣC_i=41	14	3	3 (0%) / 0 / 3	5 (67%) / 0 / 1	27
NK=4;ΣC_i=75	20	44	46 (4.5%) / 1 / 3	46 (4.5%) / 1 / 3	65
NK=5;ΣC_i=56	18	17	17 (0%) / 2 / 4	22 (29%) / 1 / 1	52
NK=7;ΣC_i=73	34	20	20 (0%) / 4 / 11	25 (25%) / 6 / 2	58
NK=8;ΣC_i=80	43	18	19 (5%) / 7 / 4	22 (2%) / 9 / 4	75
NK=10;ΣC_i=92	40	32	32 (0%) / 7 / 5	38 (19%) / 10 / 5	92
NK=12;ΣC_i=111	48	43	44 (2%) / 4 / 5	62 (44%) / 7 / 2	105
NK=16;ΣC_i=131	64	46	48 (4%) / 10 / 5	50 (9%) / 15 / 6	129
MPEG2: NK=7;ΣC_i=70	28	21	25 (19%) / 0 / 2	30 (43%) / 0 / 3	68

Table 1. Context Selection and Allocation Experimental results.

Experimental data	$TAOC$	Context loads: *Heuristic/Optimal*		
		$TAOC^R = TAOC$	$TAOC^R = TAOC^C$	$TAOC^C = TAOC$
NK=4;ΣC_i^R=53;ΣC_i^C=49	17	41 / 40	42 / 40	40 / 40
NK=5;ΣC_i^R=61;ΣC_i^C=50	26	40 / 40	40 / 40	40 / 40
NK=6;ΣC_i^R=59;ΣC_i^C=77	32	62 / 61	63 / 61	63 / 61
NK=7;ΣC_i^R=70;ΣC_i^C=95	37	92 / 87	90 / 88	89 / 88
NK=12; ΣC_i^R=122;ΣC_i^C=141	97	159 / 147	160 / 148	163 / 146

Table 2. *TAOC* row-column distribution results.

the context memory, so that context words that belong to the same block are in consecutive positions. The final solution for our example is shown in Figure 9. All the blocks, except "*f*", are placed in consecutive positions. Sometimes, the optimal solution implies some fragmentation. This usually happens when (as in Figure 9) the *TAOC* distribution among the kernels is not uniform. If it were uniform, the sizes of the dynamic blocks would be similar and, therefore, fragmentation could be avoided.

5. Experimental results

The experimental results that we have obtained (Tables 1 and 2) show that the heuristic context selection can find a nearly optimal solution in most cases and, moreover, it only takes a few iterations. The optimal values are provided through the exhaustive exploration of the mathematical model, and are only used to validate our heuristics. However, this exhaustive method can not be used for practical purposes, due to its huge exploration time. Finally, we provide the results of a random generation of feasible solutions in order to show the loss of performance that a careless scheduling may produce.

We have used synthetic experiments, as well as a practical case, MPEG2. In all the cases, a very important conclusion may be drawn from the results. The quality of the solutions obtained strongly supports the assumption of the kernel scheduler, which estimates that the context scheduling is performed in an optimal way.

Table 2 illustrates that there is no dependence between the total number of context loadings and the *TAOC* distribution between row and column positions of CM.

Finally, the context allocation finds the solution with lowest fragmentation. The fragmentation is always the lowest for the uniform $TAOC_i$ distribution, and it grows with the total number of kernels.

6. Conclusions

In this work, we presented a general strategy to face the scheduling problem in reconfigurable computing. In particular, we have focused on the context loading problem for the MorphoSys architecture. The problem is divided into two tasks: context selection and allocation. We proposed a mathematical model for context selection and two different ways to solve it. The first one is a heuristic technique for context selection, which provides a nearly optimal solution in only a few iterations. The second alternative is used to validate the heuristics through the exhaustive exploration of the feasible solutions. The comparison of the experimental results shows the good quality solutions that can be obtained. Finally, we have solved the context allocation through the exploration of a block dependency graph, so that the memory fragmentation is minimized. Future work will address data scheduling.

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, pp.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, pp. 389-396, 1998.

[6] M. Kaul and R. Vemuri, "Temporal Partitioning Combined with Design Space Exploration for Latency Minimization of Run-Time Reconfigured Designs", Proceedings of the DATE, pp. 202-209, 1999.

[7] R. Maestre, F. J. Kurdahi, N. Bagherzadeh, H. Singh, R. Hermida, M. Fernandez, "Kernel Scheduling in Reconfigurable Computing", DATE Proceedings, pp. 90-96, 1999.

[8] 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.