A Data Scheduler for Multi-Context Reconfigurable Architectures

Marcos Sanchez-Elez, Milagros Fernández, Roman Hermida, Rafael Maestre
Dept. de Arquitectura de Computadores y Automatica
Universidad Complutense
28040 Madrid, SPAIN
Tel: (+34) 91 394 70 70
marcos@eucmos.sim.ucm.es
mila45 I rhemida I maestre@dacya.ucm.es

Fadi Kurdahi, Nader Bagherzadeh
Dept. of Electrical and Computer Engineering
University of California, Irvine.
California 92697, USA
Tel: (+1) 949 824 8104
kurdahi I nader@ece.uci.edu

ABSTRACT
In this paper, we present an approach to the problem of data scheduling for multi-context reconfigurable architectures targeting DSP applications. The main goal is to improve applications execution time, through the integration of the data scheduler within a compilation framework specifically conceived for these architectures. Some amount of on-chip data storage is assumed to be available in the reconfigurable architecture. Therefore the data scheduler tries to optimally exploit this storage, saving data 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 operations required to implement the dynamic reconfiguration of the system.

1. INTRODUCTION
Reconfigurable computing systems are systems that combine programmable hardware with programmable processors. They are an alternative for implementing a wide range of computationally intensive applications such as DSP and multimedia. The most common devices used for reconfigurable computing are FPGAs [1]. However, the reconfiguration time has a negative effect on system performance. In this respect, dynamic reconfiguration has emerged as a particularly attractive technique for minimizing this negative effect [2].

An example of dynamic reconfiguration are multi-context architectures, which may store a set of different configuration planes (contexts) in a context memory. A context is a complete configuration for the entire reconfigurable chip. When a new configuration is needed, it is downloaded from the context memory, if available. As the context memory is on chip, this operation is faster than the reconfiguration from the external memory.

Previous work [3][4] discussed scheduling for multi-context architectures, in particular with MorphoSys [5] as the target architecture. This architecture has been developed for applications with a considerable amount of potential parallelism such as multimedia and DSP, which are typically composed of a group of macro-tasks that are repeatedly executed within a loop. We will use the term kernel to refer to one of these macro-tasks. At the abstraction level on which we are working a kernel is characterized by its contexts as well as its input and output data. A kernel scheduling technique was proposed in [6] to generate a kernel sequence that estimates the execution time through 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 has not been dealt with in detail previously, however it has an important influence on system performance. Every time a kernel sequence (generated by kernel scheduling) is executed, its data and results have to be transferred from/to the external memory. An improvement in data scheduling decreases data transfers. Moreover, as will be shown below, an optimal management of the internal memory can increase context reuse. Therefore, a good data scheduler can achieve a decrease in overall execution time.

Data scheduling in reconfigurable computing systems has not been discussed in detail by other authors. In [7] a data scheduler for dynamic architectures is proposed, though it does not optimize memory management. Data scheduling is addressed in the memory management area for general purpose processors [8]. Unlike these cases, many multimedia applications are periodic and their input data could be statically considered.

The paper begins with an overview of MorphoSys and its compilation framework. Section III intuitively describes the problem. In section IV we discuss the fundamental aspects of data management within a cluster (subset of kernels of the whole application), and present a new methodology to deal with this issue. In section V we show how data reuse among clusters reduces data transfers. Experimental results are presented in section VI. Finally, we present some conclusions from our research in section VII.
2. ARCHITECTURE AND FRAMEWORK OVERVIEW

This section briefly describes the target system M1, which is the first implementation of MorphoSys. We also present the development framework that integrates the data scheduling methodology presented in this paper with other compilation tasks. MorphoSys (Figure 1) is composed of an 8x8 array of reconfigurable cells (RC). Its functionality and interconnection network are configured through 32-bit context words, which are stored in a Context Memory (CM). RC array input data and results are stored in the frame buffer (FB) which is an internal data memory. The DMA controller establishes the bridge that connects the external memory to either the FB or the CM. Thus simultaneous transfers of data and contexts are not possible. The MorphoSys system operation is controlled by a RISC processor.

The data scheduler optimizes FB utilization. Organizing FB into two sets allows overlapping of computation and data transfers. One of the two sets provides computation data for the RC Array and also stores processed data and results, whereas the other set stores results in the external memory through the DMA controller and reloads data for the next round of computations. The data scheduler reduces data transfers to each set and optimizes data and results allocation.

The data scheduler has been developed for MorphoSys architecture. However, it can be applied to any system that allows overlapping of computation and data transfers between an external memory and an internal memory organized into two sets. An overview of the proposed framework is shown in Figure 2. 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 is done only once. The information extractor generates the information needed by the compilation tasks that follow it, including kernel execution times, data reuse among kernels, as well as data and context sizes for each kernel.

The kernel scheduler [6] explores the design space to find a sequence of kernels that minimizes the execution time. The kernel scheduler decides which is the best sequence of kernels and performs 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. If we suppose an application is composed of kernels k1, k2, k3, and k4, the kernel scheduler could assign k1 to one FB set, and k2, k3 and k4 to the other set. This would imply the existence of two clusters (k1) and (k2, k3, k4). While the first cluster is being executed using data of one FB set, the data and contexts of the other cluster are being transferred to the other FB set and CM.

The kernel scheduler generates one kernel sequence that minimizes the overall execution time, estimating data and context transfers. The context scheduler and data scheduler specify how and when each transfer is performed.

3. PROBLEM OVERVIEW

We propose a methodology to perform data scheduling on a given ordered set of clusters.

The problem could be defined as: Given an ordered set of clusters and its data and results sizes, find the data scheduling that minimizes the overall execution time. The target reconfigurable system allows overlapping contexts and data transfers with system computation. Therefore execution time for cluster Cc is:

\[ ET(C_c) = \text{MAX} \left( \sum_{i=1}^{n} K_i, \sum_{i=1}^{n} (D_i + R_i) + t(C_c) \right) \]

Where \( K_i \) stands for computation time, \( D_i \) for data loading time, \( R_i \) for results storing time, for kernel i of \( C_c \) and \( t(C_c) \) for \( C_c \) contexts loading time.

The data scheduler cannot minimize computation time, it can only decrease data and context loading time from external memory and results storing time to external memory.

Then the time for context and data transfers for \( C_c \) is:

\[ TT(C_c) = \sum_{i=1}^{n} (D_i + R_i) + t(C_c) \]

We can improve data management within a cluster or among clusters because:

- An optimal data and results management within a cluster improves FB use. It allows the number of consecutive iterations of kernel execution (RF) to be increased, as will be shown in section IV. If one kernel is consecutively executed RF times, its contexts have to be loaded only once. Therefore, RF is defined as the Context Reuse Factor. The data scheduler minimizes context transfers, and thus, execution time.

- An optimal data management among clusters reduces data transfers with the external memory. If a cluster data are later used by other clusters and kept in FB then reloading of these data is avoided.
4. DATA MANAGEMENT WITHIN A CLUSTER

The first step towards optimizing data scheduling is to manage data within a cluster. The kernel scheduler generates clusters whose data fit into one FB set. Furthermore, before cluster execution begins, data have to be transferred to FB. For this reason, we may consider clusters as data management units and we begin with optimizing data management within a cluster.

Data management computes the maximum RF value such the cluster data and results fit into one FB set.

4.1 Data Size

A basic data scheduling within a cluster was proposed by the kernel scheduler [6].

We consider that all cluster data have to be loaded before cluster execution begins. DS(Cc) is the maximum data size used by Cc. This may be obtained by summing all input data and results generated by all the kernels in the cluster. When a kernel has already been executed, data that are not used by the following kernels can be replaced by generated results.

Let Cc = {k1, k2, ..., kn} be a cluster, then the data size may be expressed as:

$$DS(C_c) = \max_{k \in \{k_1, ..., k_n\}} \left( \sum_{i=1}^{n} d_i + \sum_{j=1}^{g} r_j \right)$$

- $d_i$ = size of input data for kernel $k_i$ except those shared with kernels executed later ($k_{i+1}, ..., k_n$).
- $r_j$ = size of results of kernel $k_j$ that will be used as data by kernels of clusters executed later or as results of the application.

The kernel scheduler generates a cluster sequence and a kernel sequence within a cluster. The data scheduler can reorder this kernel sequence (when data and control dependencies allow it) to find the sequence that minimizes DS(Cc).

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 applications 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, so that every kernel can process the total amount of data before executing the next one. The number of consecutive executions of one kernel (RF) is limited by the internal memory size. In this case their contexts are only loaded n/RF times, achieving a decrease in overall execution time.

This data scheduling maximizes free space in the FB, that could be used to store all the cluster data for RF consecutive iterations. Data and results have the same size regardless of the iteration. In order to find data size, data used or produced are multiplied by RF (number of consecutive iterations) for all kernels in the previous expression, except one of them, say $k_i$. If $k_i$ is going to be executed RF times, the first execution of $k_i$ requires a data size of $RF \cdot d_i + RF \cdot r_{ij} + rout_j + \sum_{j=1}^{n} r_{ij}$, the second one $RF \cdot 2 \cdot Rout_j + 2 \cdot Rout_j + 2 \cdot \sum_{j=1}^{n} r_{ij}$, and so on.

Therefore, the maximum value obtained is the true data size used by $k_i$:

$$SZ(i) = \max_{k \in \{k_1, ..., k_n\}} \left( RF \cdot (d_i + \sum_{j=i}^{n} r_{ij}) + Rout_j + \sum_{j=i+1}^{n} r_{ij} \right)$$

In addition, DS(Cc) may be expressed as:

$$DS(C_c) = \max_{k \in \{k_1, ..., k_n\}} \left( \sum_{i=1}^{n} d_i + \sum_{j=1}^{g} Rout_j + \sum_{i=1}^{g} r_{ij} \right) + SZ(i)$$

By solving the previous equation, the data scheduler finds the maximum RF value that fulfills DS(Cc) ≤ FBS, for the cluster that has the maximum amount of data and results. RF may be the same for all clusters on account of data and results dependencies.

The data scheduler improves data management within a cluster by considering result reuse. It achieves the maximum available free space and, therefore, the maximum RF value. However, results reuse makes data and results assignment complex.
4.1 Data and results allocation

Data scheduling is static because data and results sizes are known at computation time and it is periodical in time. Therefore data scheduling has to be performed only once.

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

All input data of the cluster have to be loaded into FB at the same time before its execution starts. To minimize fragmentation, the data scheduler allocates to upper free addresses those data that will remain in memory longer.

First the procedure (Algorithm I) Data_allocate places data following the first-fit algorithm from address to lower addresses. Then results are placed following the procedure result_allocate from address to upper addresses.

The data scheduler starts placing at the lowest address the results of kernel kernel that are going to remain longest in memory. Then it places the next result that is going to remain longest. The process is repeated for the non-allocating results.

To make addressing easier, data and results allocation should be periodical. The data scheduler stores in the FB data of all kernels in the application for RF consecutive iterations. To maintain periodicity, data and results are allocated from the address where previous data or results were addressed (Figure 4).

The procedure release() adds to FB_list() the free space generated by results and data no longer used when one kernel iteration finishes.

Sometimes a result does not fit into any free block, so to improve memory use data_allocate and result_allocate split it

*Kernel_list*(c);

OrderedList kernel list to cluster c

*FB_list*();

Free space list into FB

kernel = last kernel in Kernel_list;

data_result_allocator{

while kernel ≠ 0 {

if k=0 there are not any kernel

for(iteration=1;iteration++;iteration<RF){

if(iteration=1) then address=upper free address

address=data_allocate(kernel,address,FB_list())

data(kernel,iteration)=address;

Load data position.

}

kernel=first kernel in Ker_list;

while k ≠ 0 /

If k=0 there are not any kernel

q=last kernel in Ker_list(c);

for(iteration=1;iteration++;iteration<RF){

if(iteration=1) then address=lower free address

address=result_allocate(kernel,address,FB_list())

This places output result

while q ≠ k/

address(q)=result_allocate(kernel,address(q),FB_list())

result(kernel,q,iteration)=address(q);

release(kernel,iteration)

kernel=next kernel in Kernel_list(c)
}
}

Algorithm I

<table>
<thead>
<tr>
<th>d_1</th>
<th>d_2</th>
<th>d_3</th>
<th>d_4</th>
<th>d_5</th>
<th>d_6</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_1</td>
<td>d_2</td>
<td>d_3</td>
<td>d_4</td>
<td>d_5</td>
<td>d_6</td>
</tr>
<tr>
<td>d_1</td>
<td>d_2</td>
<td>d_3</td>
<td>d_4</td>
<td>d_5</td>
<td>d_6</td>
</tr>
<tr>
<td>d_1</td>
<td>d_2</td>
<td>d_3</td>
<td>d_4</td>
<td>d_5</td>
<td>d_6</td>
</tr>
<tr>
<td>d_1</td>
<td>d_2</td>
<td>d_3</td>
<td>d_4</td>
<td>d_5</td>
<td>d_6</td>
</tr>
<tr>
<td>d_1</td>
<td>d_2</td>
<td>d_3</td>
<td>d_4</td>
<td>d_5</td>
<td>d_6</td>
</tr>
</tbody>
</table>

Figure 4. Data scheduling with RF = 3

into two or more parts, and as a consequence the access to this result is more complex.

This algorithm achieves a periodical data and results allocation with low memory fragmentation. Figure 4 shows figure 3 example for RF=3.

5. DATA REUSE AMONG CLUSTERS.

We can improve data scheduler because there might be free space in some clusters. This may happen because the cluster with the highest data size determines the maximum value of RF. This fact implies that the available memory during execution of the others clusters is certainly higher than that which is available when executing the cluster that has the highest data size. This free space can be used to keep data shared with following clusters, which are assigned to the same FB set. Thus these data will not be reloaded. Therefore, as many data are not reloaded, data transfers from external memory are reduced, which reduces the overall execution time.

The data scheduler finds n_c which is the number of clusters assigned to the same FB set that can share data. It starts checking if shared data with the next cluster fit in FB (DS(C_i) ≤ FBs). Then if there is still free space in FB, it checks the data shared with the following cluster, and so on, until there is not enough free space to store shared data with more clusters.

Data management among clusters entails computations of the maximum available memory space and the allocation of the shared data into this space. As FB is divided into two sets, data scheduler can manage both independently.

Let \{C_1, C_2, ..., C_n\} be a set of clusters of our application, where \{C_1, C_{n-1}, C_n\} are assigned to one FB set and \{C_2, C_{n-2}, C_{n-1}\} are assigned to the other set. Then the maximum data size for a cluster C_i considering data shared among n_c clusters may be expressed as:
DS(\(C_j\)) = \max_{k \in [1 \ldots \#C]} \left( \sum_{j=1}^{\#C} RF \cdot d_j - \sum_{j=1}^{\#C} RF \cdot D_{j, c, v} \right) + \\
+ \sum_{j=1}^{\#C} RF \cdot \text{rout}_j + \sum_{j=1}^{\#C} RF \cdot \text{r}_j \\
+ \max \left[ RF \left( d_i - \sum_{j=1}^{\#C} \sum_{v=1}^{\#v} D_{j, c, v} \right) + \text{rout} + \sum_{j=1}^{\#C} \text{r}_j \right] \right) + \sum_{j=1}^{\#C} RF \cdot D_{j, c, v}

D_{j, c, v} is the size of shared data with clusters j, ..., v, that are assigned to the same FB set. As only data shared by \(n_c\) fit into the FB then some of them must belong to \(\{C_{c+2}, ..., C_{c+(2n-2)}, C_{c+2n}\}\). And there are some \(b > c\) and \(a \leq c\) that fulfill \(b - a \leq 2n\), therefore, there are no data shared among clusters more than \(2n\), which is the maximum clusters distance for their data to fit into one FB set. Thereby, \(\sum_{j=1}^{\#C} D_{j, c, v}\) are the shared data among clusters that were stored in FB from cluster \(C_j\). These data will be released when the last cluster which uses these data \(C_c\) has been executed.

\(D_{j, c, v}\) is the size of input data for kernel \(k\) of cluster \(c\) shared with cluster \(j\), ..., \(c\), ..., \(v\) (which are not data for following kernels), that are assigned to the same FB set. As only data shared by \(n_c\) clusters fit into the FB, some of them belong to \(\{C_{c+2}, C_{c+2n}, ..., C_{c-2}, C_{c-2n}, ..., C_{c+2n}\}\). As \(d_i\) is the total data used by \(k\) and data shared among clusters do not release FB space until the last cluster that uses them is executed, then \(d_i - \sum_{j=1}^{\#C} D_{j, c, v}\) gives us the space that will be released when \(k\) execution ends.

The algorithm first allocates data of one cluster, that are shared with the next furthest cluster, and repeats the process for the rest of the clusters.

The procedure \(\text{cluster_data_shared}(c, v, RF)\) places data shared between cluster \(c\) and cluster \(v\) for the iteration \(RF\).

After shared data among clusters are placed, the data scheduler allocates data that belong only to cluster \(c\), following the algorithm explained in section IV. The procedure \(\text{data_result_allocate}\) performs the placement for the kernels.

Clusters list();
FB list();
nc;
\(c=\text{first cluster in CL list};\)
while \(c \neq 0\) {
  \(v=\text{nc cluster in CL list then } c;\)
  while \(v > c\) {
    \text{cluster_data_shared}(c, v, RF);
    \(v=\text{previous cluster in CL list};\)
  }
data_result_allocate(c);
\(c=\text{next cluster in CL list};\)
}

Algorithm II.

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.

The experiments differ in data dependencies, number of kernels and kernel information as shown in Table 1. All data sizes are in bytes.

Finding the execution time as was explained in section III, for the sequence that minimizes data and results space, we can compare the basic scheduler (developing the method suggested in the kernel scheduler to find RF) and the data scheduler with the basic scheduler (without RF), as shown in figure 5. Where the first column is the basic scheduler with context reuse and the second is the data scheduler.

As shown, the proposed methodology always finds a good solution, minimizing overall execution time and maximizing the reuse factor.

The kernel scheduling depends on data and context sizes and available internal memory. We analyze different kernel scheduler for MPEG and ATR as shown in Table 1. If a cluster contains few kernels, data reuse among clusters is improved. However, if a cluster contains many kernels, this improves data and results reuse within a cluster. The data scheduler reuses data among clusters and data and results within a cluster, maximizing the RF value as shown in Table 1.

<table>
<thead>
<tr>
<th>Table 1. Experimental results.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N: total number of cluster; n: maximum number of kernels; d: input data size per iteration; D: shared data size per iteration; r: kernel result size per iteration; (r_{\text{out}}): output result size per iteration; n: number of cluster that can share data; RF: number of consecutive iterations: RF, with basic scheduler, RF, with data scheduler).</td>
</tr>
<tr>
<td>(N)</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>E1</td>
</tr>
<tr>
<td>E2</td>
</tr>
<tr>
<td>E3</td>
</tr>
<tr>
<td>MPEG</td>
</tr>
<tr>
<td>ATR -FL</td>
</tr>
<tr>
<td>ATR -FL</td>
</tr>
<tr>
<td>ATR -SLD</td>
</tr>
</tbody>
</table>

181
7. CONCLUSION
In this paper we have shown that using a suitable strategy to minimize data and context transfers contributes to the reduction of the execution time in multi-context reconfigurable architectures. The proposed data scheduler enables the reuse of results within a cluster, releasing results and data that will not be used later. As this maximizes the available free space in the FB, the data scheduler can increase the number of consecutive iterations (RF). As a consequence, kernel contexts are reused for an increased number of iterations, so reducing context reloading and minimizing execution time. Data reuse is not limited to operations within a cluster. If the FB has sufficient free space, data to be used by other clusters can be retained, so allowing further reduction of data transfers and execution time. The work of the data scheduler is based on a specific data placement policy that tries to simplify accesses to the FB, as well as promote periodicity in both time and space. Future work will address the reuse of results among clusters. However, this possibility may complicate the placement policy. Alternative placement policies have to be studied more in depth.

8. ACKNOWLEDGMENTS
This research is supported by the Spanish Grant CICYT TIC 99-0474, and the Commission for Scientific Exchange between USA and Spain.

9. REFERENCES