Mapping and Scheduling of Directed Acyclic Graphs on An FPFA Tile

Yuanqing Guo Gerard J.M. Smit
University of Twente, Department of Computer Science
P.O. Box 217, 7500AE Enschede, The Netherlands
Phone: +31 (0)53 4894178 Fax: +31 (0)53 4894590
E-mail: {yguo, smit}@cs.utwente.nl

Abstract—**An architecture for a hand-held multimedia device requires components that are energy-efficient, flexible, and provide high performance. In the CHAMELEON [4] project we develop a coarse grained reconfigurable device for DSP-like algorithms, the so-called Field Programmable Function Array (FPFA). The FPFA devices are reminiscent to FPGAs, but with a matrix of Processing Parts (PP) instead of CLBs. The design of the FPFA focuses on: (1) Keeping each PP small to maximize the number of PPs that can fit on a chip; (2) Providing sufficient flexibility; (3) Low energy consumption; (4) Exploiting the maximum amount of parallelism; (5) A strong support tool for FPFA-based applications. The challenge in providing compiler support for the FPFA-based design stems from the flexibility of the FPFA structure. If we do not use the characteristics of the FPFA structure properly, the advantages of an FPFA may become its disadvantages. The GECKO[1]project focuses on this problem. In this paper, we present a mapping and scheduling scheme for applications running on one FPFA tile. Applications are written in C and C code is translated to a Directed Acyclic Graphs (DAG) [4]. This scheme can map a DAG directly onto the reconfigurable PPs of an FPFA tile. It tries to achieve low power consumption by exploiting locality of reference and high performance by exploiting maximum parallelism.**

Keywords—**FPFA, Directed Acyclic Graph, Parallel Compiler**

I. INTRODUCTION

The emergence of high capacity reconfigurable devices is igniting a revolution in general-purpose processing. Reconfigurable computing systems combine programmable hardware with programmable processors to capitalize on the strengths of hardware and software. Today, the most common devices used for reconfigurable computing are Field Programmable Gate Arrays (PFGAs). FPGAs present the abstraction of gate arrays allowing developers to manipulate flip-flops, small amounts of memory, and logic gates. Although less efficient than ASICs (Application Specific Integrated Circuits), reconfigurable devices, such as FPGAs can be used for implementing customized circuits. Besides ASICs and FPGAs, an architecture for a hand-held multimedia device should also contain devices that are energy-efficient, flexible and course grained. The latter can be achieved by a reconfigurable device that provides these properties for a specific application domain. Field Programmable Function Array (FPFA) developed in the CHAMELEON project, is such a device in the Digital Signal Processing (DSP) domain. The FPFA devices are reminiscent to FPGAs, but with a matrix of Processing Parts (PP) instead of CLBs (Configurable Logic Blocks). The general philosophy is to build an architecture based on replicating a simple processing part (PP), each with its own instruction stream. The focus is on keeping each PP small to maximize the number of PPs that can fit on a chip, improving the chip's clock-speed, energy consumption and the amount of parallelism it can exploit. Low power is mainly achieved by exploiting locality of reference. High performance is obtained by exploiting parallelism.

Only when the FPFA structure is used properly the strong points can be embodied, which is done during the application design period. Otherwise the advantages of FPFA can become its disadvantages. On the other hand, fully implementing the characteristics of FPFA is not a trivial matter.

An urgent necessity has arisen for support tool development to automate the design process and achieve optimal exploitation of the architectural features of the system. In the GECKO project, an FPFA oriented compiler is being developed to do this job. The compiler is capable of implementing programs written in a high-level language, such as C/C++ or Java, directly into an array of reconfigurable hardware modules on a single chip. This is done in two steps: Firstly the high-level language is transformed into control data flow graph (CDFG); secondly this control data flow graph is mapped to FPFA. In this paper only the second step is considered, and a kind of simpler CDFGs,

[1]This research is supported by the Dutch organization for Scientific Research NWO, the Dutch Ministry of Economic Affaires and the technology foundation STW.

Directed Acyclic Graph (DAG) is employed.

In Section II the structure and characteristics of the FPFA will be introduced. Section III gives the general idea of the FPFA mapping. Section IV, V and VI will deal with the scheduling and mapping scheme in detail. Section VII provides some simulation results and finally the conclusions are given in section VIII.

II. The FPFA structure

Many DSP-like algorithms (like FIR and FFT) call for a word level reconfigurable data-path. In the CHAMELEON project, a word-level reconfigurable data-path is defined, the FPFA. It consists of multiple interconnected processor tiles (see Figure 1). Within a tile multiple data streams can be processed in parallel. Multiple processors can coexist in parallel on different tiles.

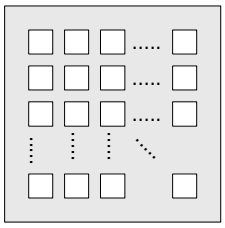


Fig. 1. FPFA architecture

An FPFA processor tile consists of five identical PPs, which share a control unit (see Figure 2). An individual

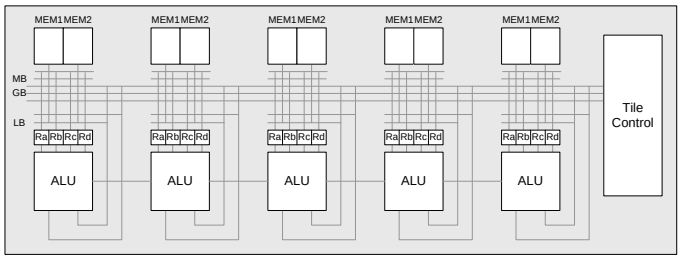


Fig. 2. Processor tile

PP contains an arithmetic and logic unit (ALU), four input register banks named Ra, Rb, Rc, Rd and two memories called MEM1 and MEM2. Each register bank consists of four registers. Each memory has 512 entries. A crossbar-switch makes flexible routing between the ALUs, registers and memories possible. The crossbar enables an ALU to write back their result to any register or memory within a tile. An ALU can only use the data from its local registers as inputs.

The execution time on the data-path is a very important factor for executing speed, which can be summarized in Table I. We see that the data in REGs can not be moved to other REGs or MEMs directly. If such kind of movements are needed, we can let the variables pass local ALUs first, then the results of the ALUs are put to proper storage places.

Path	Number of clock cycles
MEM $\overset{crossbar}{\longrightarrow}$ REG	1
MEM $\overset{crossbar}{\longrightarrow}$ MEM	1
REG $\longrightarrow$ MEM	not allowed
REG $\longrightarrow$ REG	not allowed
REG $\longrightarrow$ ALU_{idle} $\overset{crossbar}{\longrightarrow}$ REG	1
REG $\longrightarrow$ ALU_{idle} $\overset{crossbar}{\longrightarrow}$ MEM	1

TABLE I
Execution time on data-path

The main architectural issues of the FPFA that are relevant for scheduling and mapping are summarized as following:

- The size of a memory is 512.
- Each register bank includes 4 registers.
- Only one datum can be read from or written from a memory within one clock cycle.
- The crossbar has limited number of buses.
- The execution time of the data-path is fixed.

III. General Framework of Mapping and Scheduling

In this paper, we take a Directed Acyclic Graph (DAG) as the input. The translation of high level language statements (c/c++) to a control data flow graph (CDFG) is introduced in [4]. In this paper we restrict our graph to DAGs.

Before trying to find a good support tool, we need to answer one question first: what a good solution is. The answer goes back to the motivation of the FPFA design – speed and energy efficiency. High speed means that an FPFA device will execute an application programme within as few as possible clock cycles. Multiple PPs bring the possibility of exploiting the parallelism for this goal. Energy efficiency is implemented by locality of reference. However, sometimes certain conflicts between the reduction of clock cycles and locality of reference can appear, i.e., the mapping and scheduling scheme with locality of reference takes longer execution time. We choose the number of clock cycles as a more important criterion mainly because performance is a more important concern in most cases.

The problem of scheduling DAGs on an FPFA tile looks similar to the task scheduling problem on multiprocessors.

Both structures have multi-functional units for task executions. The solutions to both problems are taking speed as one of their evaluation criteria. These similarities give us the possibility to borrow some ideas from the task scheduling problem of multiprocessors systems. Kim and Browne [8] considered linear clustering which is an important special case of clustering. Sarkar [3] presents a clustering algorithm based on a scheduling algorithm on unbounded number of processors. Wu and Gajski [7] developed a programming aid for hypercube architectures using scheduling techniques. Yang and Gerasoulis [6] have proposed a fast and accurate heuristic algorithm, the Dominant Sequence Clustering. However, these algorithm cannot be used to the FPFA oriented scheduling problem directly. Compared with a processor in multiprocessor system, the concept of an PP in an FPFA tile has weaker meaning: (1) In our FPFA, the connections among different PPs of one FPFA tile are tighter than those between two multiprocessors; (2) There is no extra time consumption in data communication between two PPs of an FPFA tile. Actually an FPFA tile is more like one processor with 5 ALUs rather than five processors. Due to these differences the solution to task scheduling on multiprocessor will not fit the problem about task scheduling on one FPFA tile.

The procedure of mapping task graph to an FPFA tile has NP complexity just like the task scheduling problem on multiprocessors [5] system. To simplify the problem, on the basis of the two-phased decomposition of multiprocessor scheduling introduced by Sarkar [3], we build our work on three-phase decomposition:

1 Task clustering and ALU data-path mapping,

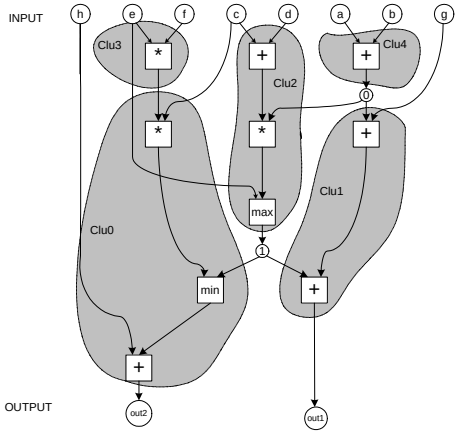
2 Scheduling the clusters on the 5 physical ALUs of an FPFA tile,

3 Resource allocation.

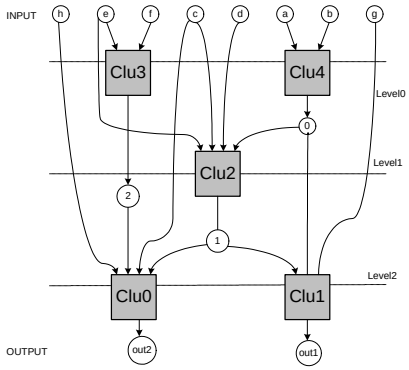
In the clustering phase we assume that the task graph is partitioned and mapped to an unbounded number of fully connected ALUs, which can perform inter-ALU communication simultaneously. This clustering and mapping scheme is based on the ALU data-path of our FPFA. The objective of the mapping is the same as the overall objective, minimization of the net execution time, i.e., finding the shortest critical path. In this phase, one design goal of FPFA, parallelism, is partly implemented. We say ”partly” because the clustering algorithm does not consider the limitation of resources. In the second and third phases the graph derived from the first phase is mapped onto the given resource-constrained architecture. In the cluster scheduling phase, the graph obtained from the clustering phase is scheduled according to the number of ALUs (in our case 5). This scheduling, together with clustering phase, reaches the goal of parallelism. In the resource allocation phase, the scheduled graph is mapped to resources where locality of reference is exploited. The separation of cluster scheduling and resource allocation will not result in too much performance loss in that the five ALUs are tightly connected. In section IV, the first phase will be introduced. Cluster scheduling is given in section V. We will address resource allocation phase in section VI.

IV. Clustering and data-path mapping

The input for clustering and data-path mapping is an acyclic directed task graph [5]. Under the assumption of an unbounded number of ALUs of a completely connected architecture, *clustering* is to put in one cluster as much computations as possible that can be executed in one clock cycle by one ALU. Goals of clustering are 1) minimization of the number of ALUs required and 2) minimization of the length of the critical path of the dataflow graph. We take the first goal as the more important one.



(a) Clustering scheme



(b) Clustering output

Fig. 3. Clustering scheme and Output

Every time we define one cluster, the possible configurations of data-path have been determined. Each configuration has fixed input and output ports, fixed function blocks and fixed control signals. The partition with one or more clusters that can not be mapped to the ALU data-path is a failed partition. For this reason the procedure of clustering should be combined with ALU data-path mapping.

Figure 3(a) gives an example of task graph where circles represent operands (values) and rectangles refer to operations (tasks). Initial input data, denoted by INPUTs, are the inputs from outside. Final output data, denoted by OUTPUTs, are the final results of an application program. Other operands are intermediate values. In Figure 3(a), there are eight INPUTs called ”a”, ”b”, ”c”, ”d”, ”e”, ”f”, ”g” and ”h”, two OUTPUTs named ”OUT1” and ”OUT2” and two intermediate nodes denoted by ”0” and ”1” respectively.

In this paper, we are not going to focus on the clustering algorithm which is postponed to be addressed in future work. Here a simple example is given to demonstrate the main concept of clustering and data path mapping. Figure 3(a) shows the cluster scheme of a DAG. The clustering result is given in Figure 3(b), where *level* is defined as such that all clusters in each level can be executed in one clock cycle.

V. Scheduling of Clusters

In a clustered graph, the longest path is referred to as *critical path*. For instance, in Figure 3(b) there are two critical paths, ”Clu4”→”Clu2”→”Clu0” and ”Clu4”→”Clu2”→”Clu1”. The optimal execution time that can be achieved by scheduling and mapping is the length of the critical path plus one, where the extra clock cycle is for loading the inputs for the clusters in the first level. If the architecture has an unbounded number of ALUs with full connectivity as the assumption in the clustering phase, a level of clustered graph can be executed within each clock cycle. Thus the best scheduling and mapping speed is achieved. However the limitation of resources prevents us from achieving this speed. The resource limitation can be divided into three categories:

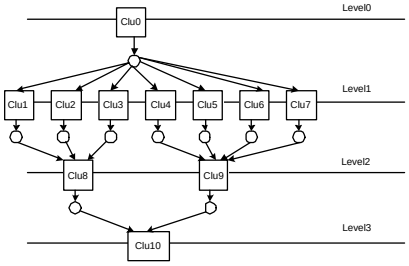
1 The number of ALUs.

2 The size of storage spaces.

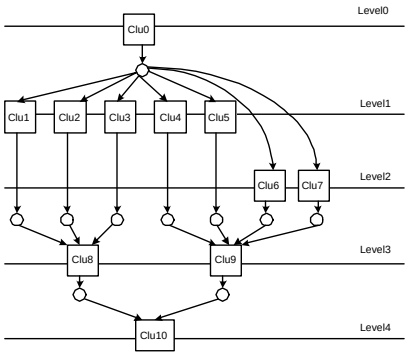
3 The limitation of crossbar and the number of writing and reading ports of memories and registers.

Item 1 and 2 are dealt with in the cluster scheduling phase. The resource allocation phase takes care of both item 2 and item 3 (see section VI).

In a clustered graph, the clusters that do not belong to any critical path can be moved up and down within the range where the dependence relations among the tasks are satisfied. In Figure 3(b), the cluster ”Clu3” is such kind of cluster. There clusters are denoted as *movable clusters*. If we move ”Clu3” from Level 0 to Level 1, the graph is still correct. If there are more than 5 clusters at some level, adjustments of some tasks could be done, but this cannot always solve the problems. For instance, if at some level more than 5 clusters are on the critical paths (see Figure 4(a)), an extra clock cycle or more have to be inserted (see Figure 4(b)).



(a) More than 5 ALUs cannot be scheduled in one level



(b) Insert a new level

Fig. 4. Insert a new level when necessary

In order to to deal with the second kind of limitation, storage size, it is desirable to store as few as possible data in the system. Still the attention is paid on the clusters that are not on the critical paths (movable clusters). We apply the following heuristics for movable clusters. If the number of released memory spaces, after execution of a cluster, is larger than the number of new generated data, the cluster should be run as soon as possible. Otherwise, that cluster should be run as late as possible. In short, as few as possible data should be kept.

After the scheduling phase, the scheduled graph satisfies: (a) There are no more than five clusters at each level; (b) The storage space needed by the system is as small as possible.

VI. Resource allocation

After scheduling, the relative executing order of clusters has been determined. However, the clusters at some level might not be executed immediately due to the limitation of resources. The resource allocation phase has to deal with this problem. The main challenge in this phase is moving of data, which is necessary for preparing inputs for ALUs and storing outputs of ALUs. The movements are arranged under the limitation of the size of register banks and memories, the number of buses and the number of reading and writing ports of memories and register banks. If all the inputs of an ALU are not prepared, the execution of the ALU is delayed as extra clock cycles are needed for data movements.

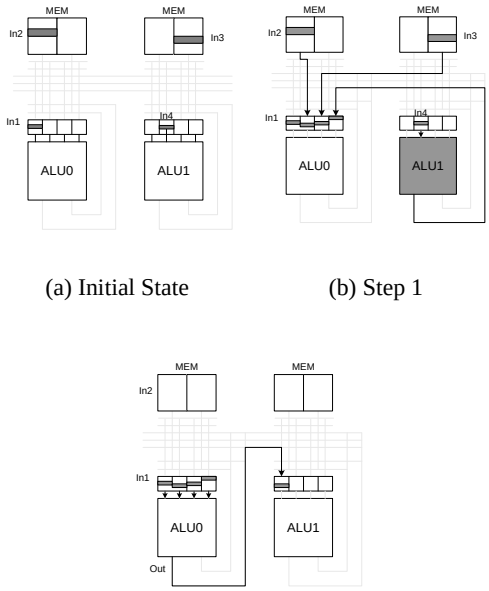
Data movements are motivated by ALU data-paths which are determined by the tasks of a cluster. Once an ALU data-path is fixed, the input register bank for an input datum is fixed, and the output port of a result datum is fixed as well. If an ALU needs an input while the input is not in the proper register bank, the datum should be moved. The source of the movement is the current address of this datum and the sink of the movement is its corresponding register bank. When a value is the computing result of an ALU, a datum movement should also be done to store this datum into a storage unit. In this case, the source of the movement is the output port of the ALU and the sink is a memory or register banks. Anyhow, the goal of resource allocation phase is to minimize the number of inserted clock cycles.

Some decisions should be made during resource allocation phase:

- Choosing memories for the initial inputs (INPUT);
- Choosing an ALU unit for each cluster;
- Defining the data movements of each clock cycle, including the information of source (memory, register, or ALU output port), sink (memory or register) and crossbar (local bus or global bus).

A. General rules of resource allocation

A good resource allocation scheme should take the whole structure of the FPFA into account. From the example shown in Figure 5, we can understand the structure of an FPFA tile and see how the characteristics of the tile affect the resource allocation. In the example, four inputs are needed by ALU0: ”In1”, ”In2”, ”In3” and ”In4”. They must be fed into ALU0 by register bank Ra, Rb, Rc and Rd respectively. This is determined by the ALU data-path mapping (see section IV). ”In1” is at the proper register bank, so it can be used by ALU0 directly. ”In2” should be fed into ALU0 through register bank Rb. Since ”In2” is located in local memory, before being used, it has to be moved from memory to register bank Rb. According to Table I, this kind of movement takes one clock cycle and needs the help of the crossbar. The local memory bus can do this job here. Unlike ”In2”, ”In3” is in the memory of ALU1. The movement needs the help of the global bus. ”In4” is staying at an improper register bank. According to Table I, REG→REG is not possible. The datum in ALU1’s register bank Rb can only be moved to ALU0’s Rd through ALU1. Compared with the situation of ”In1”, preparing ”In4” for ALU0 takes one more clock cycle and needs the help of an ALU and crossbar.



(a) Initial State (b) Step 1

(c) Step 2

Fig. 5. DifferentVariablePosition

The positions of inputs to an ALU can be a proper register, a local memory, a non-local memory or an improper register which are represented by ”In1”, ”In2”, ”In3” and ”In4” respectively. The conclusions from this example is summarized as following:

1 As ”In1”, an input staying at proper register can be used directly which is the desired situation during mapping.
2 ”In2” represents the inputs that are at the local memories relative to the ALUs that are going to use the data. The movement of such kind of data only needs the local memory bus and it consumes less energy than non-local memory. If the arrangement of case 1 is not possible, this is our second choice. Sometimes this scheme can bring some trouble as well. One reason is the memory size. Each memory has only 512 entries. When local memories are full, non-local memories have to be used. The second

reason is the writing and reading ports. Each memory has only one writing port and one reading port. One datum can be read from or written to a memory in each step. As a result, conflicts might happen between locality of reference and speed. For example, if ”In3” of Figure 5 is put in the same local memory as ”In2”, (see Figure 6), it will cost one extra step to load ”In2” and ”In3”. As already mentioned in section III, speed is a more important criterion, so it is better to put ”In3” at some non-local memory.

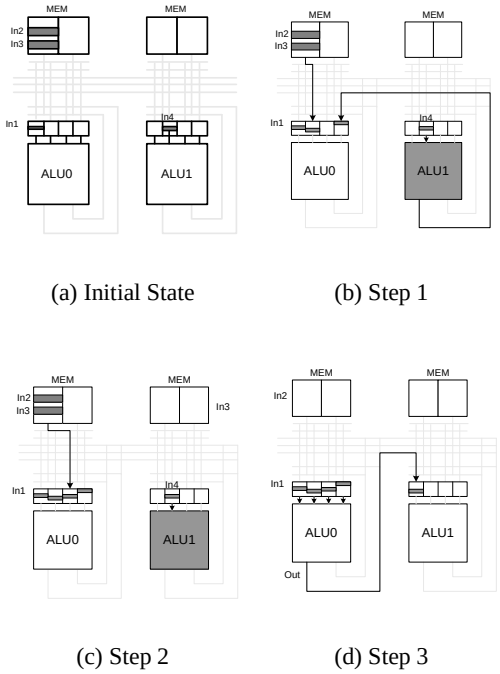


Fig. 6. Two inputs are at the same local memory.

3 When locality of reference brings lower speed, or when the local memory is full, we should try to put a datum into non-local memory like ”In3”. The movement of a data from a non-local memory to a register bank consumes more energy than from a local memory. However, as shown in Figure 5 and Figure 6, sometimes putting a datum at non-local memory is a better choice.
4 Situations like ”In4” waste energy and resources. They should be avoided.

These conclusions reflect the spirit of FPFA design – speed and locality reference. We treat them as the general principle throughout the procedure of resource allocation. Based on the general principle some rules for making each decision will be given in the following subsections.

B. Rules for controlling the lifetime of data

Due to the limitation of storage spaces, it is preferable to release the storage space as soon as possible. The starting point of an initial input’s (INPUT) lifetime is the initial step. The INPUT stays in a memory in the beginning. An intermediate value’s lifetime starts when it is computed and stored to a memory, a register or more than one place. Once a datum is moved from one storage unit to another, a decision needs to be made whether the old storage space can be released or not. The lifetime of a datum is closely related to the ALU which compute the datum (determine the starting point of the lifetime) and the last ALUs which will use the datum (determine the ending point of the lifetime). For simplicity in description, we give a name to the cluster which computes a datum ”a”, PreCluster of ”a”, and a name to the clusters which are going to use a datum ”a”, PostCluster of ”a”. An intermediate value only has one PreCluster, but it can have more PostClusters. That is the case when the datum will be used as inputs by more clusters. Among all the PostClusters, the one with the smallest level number is called the FirstPostCluster and the one with largest level number is called the LastPostCluster. In Figure 3(b), the PreCluster of datum ”0” is ”Clu4”. ”0” has two PostClusters: FirstPostCluster ”Clu2” and LastPostCluster ”Clu1”. *Vertical distance* is an concept on the scheduled clustered graph. The vertical distance of two clusters is the absolute difference between the level numbers of these two clusters on the scheduled clustered graph. Some rules regarding saving intermediate results and releasing storage spaces are as follows:

B.1 **Save an intermediate result**

The movements of an intermediate datum ”a” are determined by its storage place and the corresponding ALUs’ position of the PreCluster and PostClusters. It is better to choose the proper storage place according to the mapped positions of the PreCluster and PostClusters. There are three possible choices:

Choice 1 Put ”a” to memory first. Before execution of a PostCluster, move it to the proper register. The proper register refers to the register bank from which the datum should be fed into ALU. This is determined by data-path mapping (see section IV).
Choice 2 Put ”a” to a proper register directly.
Choice 3 Put ”a” to more than one place at the same time.

If ”a” has only one PostCluster and the vertical distance between its PostCluster and PreCluster is small, it is better to take choice 2 instead of choice 1. Firstly, choice 2 is more energy efficient than choice 1; secondly, choice 2 is faster than choice 1. Since ”a” will be used soon, if it is not ready before being used, extra clock cycles will be needed to load the datum. Therefore more attention should be paid on the speed when vertical distance between PostCluster and FirstPreALU is small.

If ”a” has only one PostCluster but the vertical distance between its PostCluster and its PreCluster is large, taking choice 2 results in ”a” staying at a register for a long period. As there are only four registers in each register bank, it is not wise to let a datum stay there for too long time. While the lower speed of choice 1 unlikely brings delay due to the large vertical distance, in this case taking choice 1 is preferable.

When ”a” has more PostClusters, and there exist PostClusters that are far from its PreCluster in vertical distance, for the same reason as described in previous paragraph, ”a” should be put into a memory. At the same time if there are some PostClusters with short vertical distances from the PreCluster, these PostClusters are treated separately, i.e., ”a” will be put into proper register banks corresponding to the PostClusters simultaneously. As in Figure 7, ”a” is the result of an ALU. It is written to different places using one global bus. If two PostClusters require ”a” to appear at the same register bank, the movement is done only once.

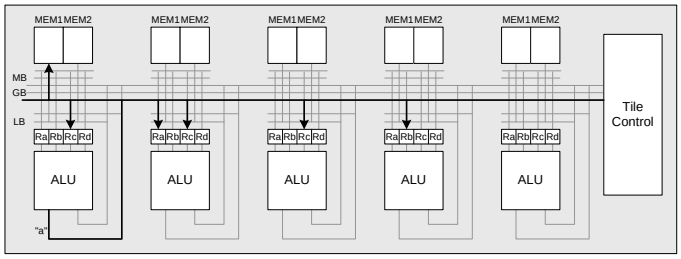


Fig. 7. Write a datum to more than one place simultaneously.

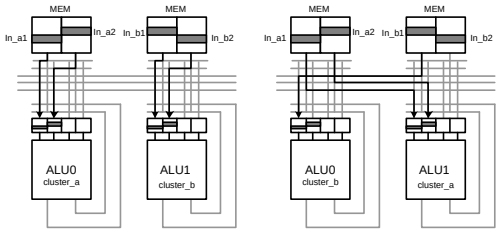
B.2 **Release storage spaces**

When a datum is moved from a memory to a register, a check should be done whether the old copy is necessary to be kept in the memory or not. The space is released when the movement to the proper register bank of each PostCluster has been arranged.

In most cases, the register keeping a datum can be released after the datum is fed into ALU. An exeption is that there is another PostCluster which shares the copy of the datum and has not been executed. This is true when the proper registers defined by two PostClusters are the same one, and the vertical distance of the two PostClusters are small.

C. Rules for allocating ALUs and INPUTs

The graph after clustering and scheduling phase contains less than five clusters at each level. There are $5 \times 4 \times 3 \times 2 \times 1 = 120$ possible solutions to assign the clusters of each level to five ALUs. A good ALU allocation scheme requires less global data movements and brings less troubles in preparing inputs for all the ALUs. Figure 8 shows the effects of different assignments on data movements. There are two clusters in a level. ”Cluster_a” takes ”In_a1” and ”In_a2”as its inputs, and ”Cluster_b” takes ”In_b1” and ”In_b2” as its inputs. According to the assignment in 8(a), all the data are local with respect to the ALU which uses them. However according to another assignment shown in Figure 8(b), no inputs are local to their ALUs. The movements of all the inputs to proper registers need the participation of global bus. The second mapping scheme, on the one hand, takes more energy consumption; One the other hand, the speed may be affected under the constraints of the limited number of global buses. In Figure 8(b), there are 3 free global buses left. The moving of ”In_b2” cannot be finished within the same clock cycle, which brings the necessity to insert an extra clock cycle.



(a) Assignment 1 (b) Assignment 2

Fig. 8. The effects of Different ALU Assignments

The outputs of one clock cycle are the inputs for the following clock cycles. ALUs allocation scheme has effect on the nature of both inputs and outputs: the movements of some inputs can change from global to local, or vice versa, so do the movements of outputs. To find the best global choice is another NP complexity problem. For simplification, we will do it in a forward way. The allocation scheme only takes care of the locality and loading difficulty of its own inputs at each step. The effects of the assignment on the following steps are not taken into account. It is up to the following steps to care about their inputs. Some negative effects from the previous steps can be deduced by permuting the ALUs of the following steps. In other words, the positions of all the inputs of a level determine the ALUs allocation of the corresponding clock cycle.

In the following part, the attention is paid to the ALUs allocation of one clock cycle which is called the current clock cycle. The corresponding level of the current clock cycle on the scheduled clustered graph is called the current level. As mentioned above, the allocation of ALUs will be done in a forward way, so we suppose the clusters at

the previous levels (the level above the current level) have already been mapped to ALUs while the executing ALUs of the clusters at the following levels are not determined yet. According to the general rules, some basic points can help us in making choice:

Case 1 As we mentioned in subsection VI-B, if an input of the current level has small vertical distance between its PreCluster and the current level, it will be saved into the proper register directly. Since ALUs have no output registers, the saving procedure has to be taken at the step where the PreCluster is executed. In this case, assigning the intermediate value's PostCluster to the same ALU as its PreCluster can make the movement of that saving procedure local.

Case 2 If the intermediate input's vertical distance between its PreCluster and current level is large, according to subsection VI-B, the datum should be moved to a memory immediately after the execution of PreCluster, and then moved to proper register before the current step. When the ALUs of current step are allocated, the storage place of the immediate value inside memory is known. According to this input, it is preferable to map its PostCluster to the ALU which belongs to the same PP as the storage memory of this input.

Case 3 If all the inputs are INPUTs or if there are several available mappings, proper ALU allocation is done on the basis of proper choice of storage places of INPUTs.

To choose memory places for "INPUT"s and intermediate data we should take into account several aspects of the system.

- Locality of reference.
- The limitation of memory space. The memories with more free space are preferable.
- Recent reading frequency. This refers to the frequency of reading data from a memory during several last clock cycles. Only one datum can be read from a memory each step. If a memory's reading port is always busy, execution of an ALU can be delayed due to the movements of some inputs. This is to avoid the situation of "In2" and "In3" in Figure 6.

All the possible 120 permutations can be checked to find the best allocation. For simplicity, the clusters can be assigned to ALUs one by one by sacrificing some performance.

D. Rules for using the crossbar

According to the functionality, the data movements through the crossbar can be divided into three groups: (a) Preparing inputs for ALUs, (b) Saving outputs of ALUs, (c) Combination of (a) and (b), as the case 1 in subsection VI-C. Saving an ALU output must be done at the clock cycle within which the output is computed. Preparing an input should be done before it is used. However, if it is prepared too early, the input will occupy the register for a too long time. A proper solution in practise is starting to prepare an input 4 clock cycles before the clock cycle it is used. If the outputs are not moved to registers or memories immediately after generated by ALUs, they will be lost. For this reason the movements in type (b) and type (c) have the priority to use the crossbar. When the resources have spare capacity left after the movements of the outputs, type (a) movements will be considered. Now the conceptions used for describing the vertical distance, "large" and "small", become clear as well. The vertical distance is "large" when it is larger than four, otherwise, it is "small".

E. The Heuristic Algorithm

The algorithm implementing all the rules described above is actually a heuristic algorithm. The pseudocode for this algorithm is listed in Figure 9. The clusters in

```
1   function ResourseAllocation(G)
2   {
3       for each level in G
4       do
5           Allocate(level);
6   }

7   function Allocate(currentLevel)
8   {
9       Allocate ALUs of the current clock cycle and put
10      INPUTs into memories;
11      for each output whose PostCluster has vertical
12          distance larger than 4
13      do
14          store the output to a memory;

15      for each input of current level
16      do
17          try to move it to proper register at the clock
18          cycle which is four steps before; If failed,
19          do it three steps before; then two steps
20          before;  one step before.

21      if  some inputs are not moved successfully
22      then insert one or more clock cycles before
23          the current one to load inputs

24  }
```

Fig. 9. Pseudocode of the heuristic resource allocation algorithm

the DAG graph are allocated level by level (lines 1-6). For each level, firstly, the ALUs and INPUTs are allocated (lines 9-10) according to the rules in subsection VI-C. Secondly, in terms of the priority of using the crossbar (see subsection VI-D), all the outputs are stored (lines 12-14) following the rules in subsection VI-B.1, and then the rest resources are used to load inputs (lines 15-20) for the following steps, where the problem on releasing storage

spaces should be considered based on the rules in subsection VI-B.2. In Figure 9, we cannot directly see the codes for saving the outputs whose PostClusters have smaller vertical distance (≤ 4) because the saving and loading procedures in this case are combined (see case 1 in subsection VI-C). Finally, extra clock cycles are inserted if necessary (lines 21-23).

VII. Simulation

We run the resource allocation algorithm on the clustered data flow graph shown by Figure 3(b). On this simple graph, there exits no level with more than five ALUs, which makes cluster scheduling an unnecessary phase. Following all the rules mentioned above, a mapping scheme with 4 decoding clock cycles and 4 times global movements comes out. For comparison, both random ALUs allocation scheme and random variable allocation scheme are tested for 40 times. Figure 10 gives the simulation results, where the number in the grid refers to the times, among the 40 tests, of the needed number of execution clock cycles and the frequency of used global buses. The blue number and the red number correspond to the random ALUs allocation scheme and the random variable allocation scheme respectively. The best solution among all the experiments is the one with 4 clock cycles and 4 global buses, which is reached by applying the rules in section VI as well.

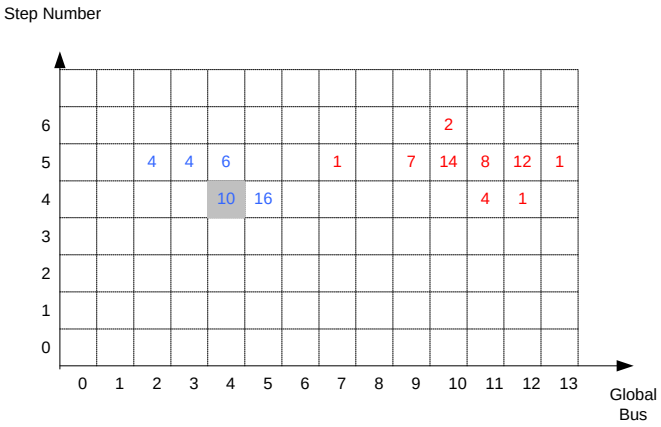


Fig. 10. Simulation Results

VIII. Conclusion

In this paper we present a scheme to mapping a CDFG to one FPFA tile. The procedure of mapping is divided into three phases: clustering, scheduling and resource allocation. Some rules for resource allocation are given by thorough analysis and summarization, the validity of which is proved by the simulation result. High performance and low power consumption are achieved by exploiting maximum parallelism and locality of reference respectively. In conclusion, using this mapping scheme the potential advantages of FPFA are fully exploited. To date, the work does not deal with the issue of CDFG with loops and branches, which will be done in the future work.

References

[1] Heysters P.M. et.al, "Exploring Energy-Effi cient Reconfi gurable Architectures for DSP Algorithms", *Proceedings PROGRESS 2000 workshop*, pp. 43-52, ISBN 90-73461-25-1, Utrecht, The Netherlands, October 2000.

[2] Jing-Chiou Liou, Michael A. Palis, "An Effi cient Task Clustering Heuristic for Scheduling DAGs on Multiprocessors", 8th Symposium on Parallel and Distributed Processing, Workshop on Resource Management in Computer Systems and Networks, New Orleans, Louisiana, October 1996.

[3] Vivek Sarkar. *Clustering and Scheduling Parallel Programs for Multiprocessors*. Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Massachusetts, 1989.

[4] Gerard J.M. Smit, Paul J.M. Havinga, Lodewijk T. Smit, Paul M. Heysters, Michel A.J. Rosien: "Dynamic Reconfi guration in Mobile Systems" accepted for publication in the 12th International Conference on Field Programmable Logic and Application (FPL2002), september 2002.

[5] Hesham EL-Rewini, Theodore Gyle Lewis, Hesham H. Ali, *Task scheduling in parallel and distributed systems*, PTR Prentice Hall, 1994.

[6] Tao Yang; Apostolos Gerasoulis, "DSC: scheduling parallel tasks on an unbounded number of processors", *IEEE Transactions on Parallel and Distributed Systems*, Volume:5 Issue:9, Sept. 1994 Page(s): 951-967.

[7] M.Y.Wu and D.Gajski, *A programming aid for hybercube architecture*, Journal of Supercomputing, 2, 1988, 349-372.

[8] S.J.Kim and J.C.Browne, A General Approach to Mapping of parallel Computation upon Multiprocessor Architectures, *International Conference on Parallel Processing*, vol 3, 1988, pp.1-8.