Exploring the Design Space of an Optimized Compiler Approach for Mesh-Like Coarse-Grained Reconfigurable Architectures

Gregory Dimitroulakos[1], Michalis D. Galanis[2] and Costas E. Goutis[3]

[123]VLSI Design Laboratory, Electrical and Computer Engineering Department, University of Patras, Patras, Greece

{dhmhgre[1],mgalanis[2],goutis[3]}@ee.upatras.gr

Abstract

In this paper we study the performance improvements and trade-offs derived from an optimized mapping approach applied on a parametric coarse grained reconfigurable array architecture. The processing elements' local register files and the processing elements' interconnection network is exploited for caching memory data values with data reuse opportunities. The data reused values are transferred through the processing elements' interconnection network hence, relieving the bus from the burden of transferring these values. A novel mapping algorithm is also proposed that uses a modulo scheduling technique. This algorithm targets on a flexible architecture template which permits experimental exploration over different architecture alternatives. The experimental results showed that the operation parallelism was significantly improved by our mapping approach. Additionally, we have outlined the relation that exists between the performance improvements and the memory access latency, the interconnection network and the processing elements' register file size.

1. Introduction

Coarse-Grained Reconfigurable Array (CGRA) architectures [1]-[4] have been proposed for accelerating computation intensive parts of algorithms residing in several scientific domains. These kinds of applications have high amounts of operation and data parallelism. The large number of PEs available in CGRAs can be used to exploit this parallelism and thus accelerating the applications' loops. CGRA architectures consist of a large number of Processing Elements (PEs) organized in a 2-Dimensional (2D) array and connected with a configurable interconnect network. This type of reconfigurable architecture is increasingly gaining interest because it is simple to be constructed and it can be scaled up, since more PEs can be added in the mesh-like interconnect. Each PE typically contains a Functional Unit (e.g. ALU, etc.), a small local memory and a configuration cache.

An increase in the operation parallelism results in a respective increase in the rate by which data are fetched from memory called -*data memory bandwidth*- which is a major bottleneck in exploiting the inherent parallelism [5]. Hence, a mapping methodology for CGRAs which tackles the memory bandwidth problem is necessary. Additionally, a study to show how the improvements, in performance, are affected from the architectural characteristics is significant as well. Such a study can provide valuable knowledge about the effect of design parameters on the quality of the resulting architecture, its performance, storage needs, and flexibility. Pursuing such a study requires a flexible architecture template, together with a resource aware scheduling technique.

In this paper a new mapping approach is proposed for mapping applications to CGRAs. It is based on a modulo scheduling technique where the scheduling, data allocation and spilling are performed in a single step. Moreover, the high bandwidth foreground memory which is constituted from the PEs' register files and the interconnections among them is exploited for the purpose of storing variables with data reuse opportunities. The data reused values are transferred through the CGRA's interconnection network instead of using the bus. Moreover, since the optimization of the memory bandwidth problem is becoming increasingly important, this work investigates experimentally how the three important architecture's parameters (memory access latency, interconnection topology and register file size) affect the efficiency of such and optimized approach in respect to the operation parallelism (ILP) and performance. For this reason the scheduling algorithm targets on a parametric architecture template having its characteristics drawn from popular CGRAs.

The rest of the paper is organized as follows: in section 2, the previous work is presented. Section 3 presents the architecture template to which the methodology targets. Section 4 describes the proposed methodology, while the experimental results are presented in section 5. Finally, conclusions are outlined in section 6.

2. Related Work

Many CGRA architectures have been proposed in the past few years [1]-[4] accompanied with an equal number of mapping methodologies. Typically the applications which belong to the application domain of CGRAs are characterized by high data transfer rate between the processor and the memory. Only a few approaches ([4],

1-4244-0054-6/06/$20.00 ©2006 IEEE

[6], [7], [8], and [9]) have been followed to tackle the problem of the limited memory bandwidth in CGRAs for exploiting the hardware parallelism.

In [6] a CGRA architecture was presented. Each PE has two input and two output registers while its operation is data-driven. The PEs' interconnection network has direct unidirectional connections among neighbouring PEs. The mapping methodology for this architecture is based on a simulated annealing algorithm in which the routing of data is considered during the operations' placement phase. For reducing the memory bandwidth requirements this mapping algorithm uses a global register file for storing frequently reused data values.

The PACT-XPP [2] is a hierarchical array of coarse-grain processing array elements. A series of vertical and horizontal buses establish communication among the PEs while for storing the intermediate data values shared memory banks exist on the left and the right side of each array's row. To reduce the number of memory accesses, the compiler [7] only reads one element per iteration and generates shift registers to store the data reuse values when array references inside loops read subsequent element positions.

In [8] a generic template for a wide range of CGRAs was presented. A three-level mapping algorithm is used to generate loop pipelines fit into the CGRA. First, on the PE-level mapping stage, microoperation trees are mapped to single PEs without the need of reconfiguration. Then the PE-level mappings are grouped together on line-level in such a way, that the number of required memory accesses not exceed the capacity of the memory interface belonging to the line. On the plane-level phase, the line-level mappings are put into the 2D array.

In [4] an array architecture is proposed in which each PE contains instruction memory data memories, an arithmetic logic unit, registers and configurable logic. The static RAM distributed across PEs alleviates the memory bandwidth bottleneck and provides shorter latency to each memory module. The placement of operations in the PEs precedes the scheduling phase as a separate step while the scheduling of operations is performed by a list scheduling algorithm.

In [9] we had proposed a list scheduling technique to reduce the data transfer bottleneck by using the local PE storage and the CGRA's interconnection network. The current work achieves better improvements with the proposed modulo scheduling technique. Moreover, it handles more effectively the register allocation problem and the routing of data values by introducing a new set of heuristics. Hence, we consider our current work as an enhanced version of our previous work.

3. CGRA architecture description

In this section, the generic reconfigurable template used for the proposed mapping methodology is described. As shown in Fig.1a it consists of 4 basic parts; (a) a 2D array of PEs connected via an interconnection network, (b) a data memory interface which includes a set of buses, a scratch-pad memory and the main memory module (c) the configuration memory and (d) the execution control unit. The design specification of the 2D array and the data memory interface is generic as to include as much as possible characteristics from existing CGRAs and parametric as to model a large number of alternatives of such architectures. Table 1 illustrates the design parameters for the considered CGRA architecture template.

Table 1. CGRA architecture design parameters

Component	Description
CGRA Interconnection Network	
structure	Pair wise connections (PE_x,PE_y)
interconnection width	2^x bits $x\varepsilon\{3, 4, 5,\ldots\}$
interconnection latency	0 cycles
PE microarchitecture	
operand bitwidth	2^x bits $x\varepsilon\{3, 4, 5,\ldots\}$
Register files' size	2^x words $x\varepsilon\{1, 2, 3,\ldots\}$
context RAM size	2^x $x\varepsilon\{1, 2, 3,\ldots\}$ configuration contexts
fast reconfiguration overhead	0 cycles
context RAM fill latency (1 configuration)	$x\varepsilon\{1, 2, 3,\ldots\}$ cycles
Operation Set	{add, multiplication, ALU}
Per Operation Latency	$x\varepsilon\{1, 2, 3,\ldots\}$ cycles
Register Files Input Ports	$x\varepsilon\{1, 2, 3,\ldots\}$ ports
Register Files Output Ports	$x\varepsilon\{3, 4, 5,\ldots\}$ ports
Data Memory Interface	
PE-Buses Connections	Pair wise connections (PE_x,Bus_y)
Scratch Pad's Access Latency	$x\varepsilon\{1, 2, 3,\ldots\}$ cycles
Number of Scratch-Pad's ports	$x\varepsilon\{1, 2, 3,\ldots\}$ ports
Main Memory Module Access latency	$x\varepsilon\{1, 2, 3,\ldots\}$ cycles
Bus Bitwidth	2^x $x\varepsilon\{3, 4, 5,\ldots\}$ bits
Number of Buses	$x\varepsilon\{1, 2, 3,\ldots\}$ buses
Bus Multiplexing	Pair wise connections (Scratch Pad's $port_x$,Bus_y)
Memories' Size	x words

The PE interconnection network is defined as a set of pairwise PE to PE connections. In more detail, depending on the PE communication network description different network configurations can be instantiated. For example, each PE can be connected only to its nearest neighbours as in [6] while there are cases, like in [10], where there are also direct connections among all PEs across a column and a row (Fig.1b). The interconnection network with the PEs' local register files acts as a high-bandwidth foreground memory, since during each cycle several data transfers can take place through different paths in the CGRA. We call it to hereafter as the Distributed Foreground Memory (DFM). Our methodology exploits this capability for reducing the memory accesses thus, reducing the data transfer bottleneck.

Additionally, each PE consists of one Functional Unit (FU), which it can be configured to perform a specific word-level operation in each cycle. The operations supported by the FU are: ALU, multiplication, and shifts. For storing intermediate values between computations and values fetched from memory, a local register file exists inside a PE. Furthermore, the FU in the proposed CGRA template supports predicate operation [11]. Predicated register files exist inside the FU for storing predicated values. Thus, loops containing conditional statements are supported by the CGRA template through the "if-conversion" process [12]. Moreover, both the predicated

register files and the local register files are rotating register files [13] (Fig.1d) for realizing the necessary register renaming mechanism for modulo scheduling. Finally, the number of input/output ports of the PE's local register files is included in the PE microarchitecture description shown in Table 1.

Fig.1d shows an example of PE architecture where it is assumed that it is connected with 4 neighbouring PEs (PE 1-4). The FU has three inputs and two outputs. The multiplexers are used to select each input operand that can come from three different sources: a) from the same PE, b) from the memory buses and c) from another PE. This PE architecture permits an operation's execution to take place without using the register files if the input operands arrive simultaneously with the operation issue time. Also, the output of each FU can be routed to other PEs through its local register files. Finally, the Context Cache stores the context words that determine how the FU, the register files, the decoders and the multiplexers are configured.

Furthermore, the CGRA's data memory interface consists of: (a) *The memory buses*. In the architecture studied in this paper the PEs residing in a row, share a common bus connection to the scratch-pad memory (Fig.1a). This also happens in existing CGRA architectures such as [8] and [10]. Furthermore, an Address Generation Unit (AGU) (Fig.1c) is attached to each bus for the addressing. Thus, the mapped loop can access a variable existing in the scratch-pad or main memory by generating the proper address.

(b) *The scratch pad memory* [14]. The scratch pad memory serves as a local memory for quickly loading data in the CGRA's PEs. However, the data values are transferred in the CGRA through the buses which have limited bandwidth if we compare them with the bandwidth requirements for executing applications to CGRAs. Thus, the bandwidth bottleneck in this type of architecture exists in the communication path that connects the CGRA and the memories. Our mapping method stores values with data reuse opportunities in the DFM. Then these variables are transferred using the internal interconnection network instead of using the bus.

(c) The main memory module can be a typical SDRAM chip.

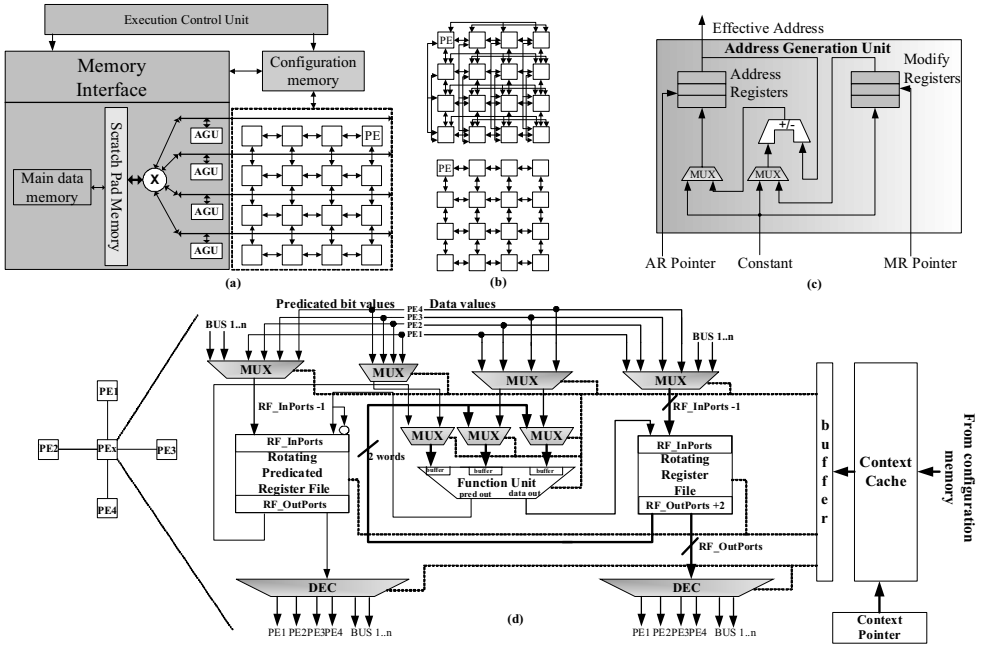

Figure 1. a) CGRA Architecture Template, b) AGU architecture, c) Examples of PE interconnection scenarios, d) PE architecture template

The configuration memory (Fig.1a) of the CGRA stores the whole configuration for setting up the CGRA for the execution of the application's critical loops. Context Caches distributed in the CGRA inside the PEs are used for the fast reconfiguration of the CGRA. The Context Cache stores a few configuration contexts locally, which can be loaded on cycle-by-cycle basis. The configuration contexts can also be loaded from the configuration memory at the cost of extra delay if the Context Cache is not large enough to store the configuration for the loop body. For transferring the configuration contexts from the configuration memory to the PEs' Context Caches a set of buses is used. Each of them can be shared either among PEs residing in the same row either in the same column. A special purpose register called Context Counter is used to point to the current configuration context. Finally, at the initialization phase the configuration contexts are loaded

from the configuration memory into the Context Caches in the same way as it happens in an FPGA device.

4. Proposed Mapping methodology

4.1. Proposed Methodology Description

Fig.2a shows the structure of the developed mapping methodology for CGRAs. The input is the application's description in C language. The first methodology step concerns the application of source level code transformations for increasing the locality of memory references as described in [15]. In this way, for a given size of DFM more data reused values can exploit it instead of using the buses. Hence, better improvements can be achieved from the mapping phase. Afterwards, the loop normalization transformation [12] is utilized for normalizing the candidate loops. Also, loop unrolling is performed for increasing the ILP in the mapping phase. Since the unlimited unrolling can lead to resource congestion situations a feedback in our methodology script refers to the exploration performed for finding the best value of the unroll factor in terms of the ILP.

For creating the code's Intermediate Representation (IR) we have utilized the front-end of the SUIF2 compiler infrastructure [16]. We have used existing and we have developed new SUIF2 passes for performing analysis and transformations on the application's loops. More specifically, data-flow analysis is used to identify live-in and live-out variables and data dependence analysis to determine the data dependencies and data reuse opportunities. Also, transformations like dead code elimination, common sub-expression elimination and if-conversion transformations have also been utilized. Moreover, to create the Data Dependence Graph (DDG) we represent the application's loop in static single assignment form to minimize the Anti- and Output dependences. The considered analysis and transformations flow are enclosed in the dashed line of Fig.2a.

Finally, the DDG of the loop body, with extra information concerning the data reuse opportunities is the application's representation for the mapping phase. We call this graph *Data Dependence Reuse Graph* (DDRG). The DDRG is generally a cyclic directed graph $G(V, E, E_R)$, where: V is the set of DDRG nodes representing the operations of the loop body. Each DDRG node is annotated with the type of operation, its priority and the memory operations it requires. E is the set of data edges showing data dependencies among operations. Each dependence edge E is annotated with the type of dependence as well as with the *dependence distance*. The *dependence distance* equals the number of iterations the dependence spans. Finally, E_R are non-directional edges showing when data reuse exists among the DDRG nodes. The E_R edges are further annotated with the names of variables that are common to the operations that connect and the *data reuse dependence distance* which equals the number of iterations between subsequent uses of the common variable. Also, we call, the subset of operations in the DDRG that have E edges sinking into a specific node v, *Data-Dependence-Predecessors* (DDPs) for that node.

The CGRA graph which is also an input to the mapping phase is an undirected graph, $G_A(V, E_I)$ where V is the set of PEs of the CGRA and E_I are the interconnections among them. The CGRA architecture description also includes the parameters, shown in Table 1.

4.2. Proposed Mapping Algorithm Fundamentals

The proposed modulo scheduling algorithm is based on the two stage hierarchical reduction technique described in [17], which can be applied to VLIW processors and with graphs with dependence cycles. According to this algorithm all loop iterations have identical schedules which are initiated at regular time intervals in a way that honours the data dependences and resources constraints. The time interval between the start of two successive iterations is called *initiation interval II* and represents the degree by which iterations overlap to each other.

Our CGRA architecture has principally 6 types of explicitly scheduled resources [18] that must be considered during the scheduling phase. These are 1) the PEs, 2) the buses, 3) the CGRA interconnections, 4) the PE's register files, 5) the register files input/output ports and 6) the AGUs. According to [18] a schedule is valid if there are no conflicts between this category of resources. To indicate the usage of a particular resource a data structure called Modulo Reservation Table (MRT) [19] is employed and it has the characteristics of a complex reservation table [19]. If scheduling an operation at some particular time involves the use of resource R at time T, then location $((T \bmod II),R)$ of the table is used to record it.

Also, the scheduling strategy followed for a CGRA architecture is different in respect to the one followed for a VLIW. This is mostly due to the routing of data values through the CGRA interconnection network. In CGRAs the operations' issue (execution) time can be determined only after they are mapped to a specific PE. This is due to the operands' routing delay (see section 4.3.1) which is generally different for mapping the operation on different PEs. Hence, the closure of dependence constraints which is calculated prior scheduling in [17] for assuring the satisfaction of dependence constraints cannot be used since the execution time for each operation is not considered fixed as explained above.

During the scheduling phase two types of constraints have to be considered for deriving a valid schedule. The first is the resource constraint which is satisfied by reserving, for executing each operation, MRT time slots which don't cause resource conflict with other already scheduled operations. The second is the data dependence constraint. This constraint is honoured by two means: 1)

An operation is ready to execute when its input operands from predecessor operations which have zero dependence (intra-iteration dependences) distance are available 2) When an operation is scheduled before its predecessor from which it is dependent with dependence distance greater than zero (inter-iteration dependences) the predecessor's output variables should be available at the successor's operation inputs at the time that is issued.

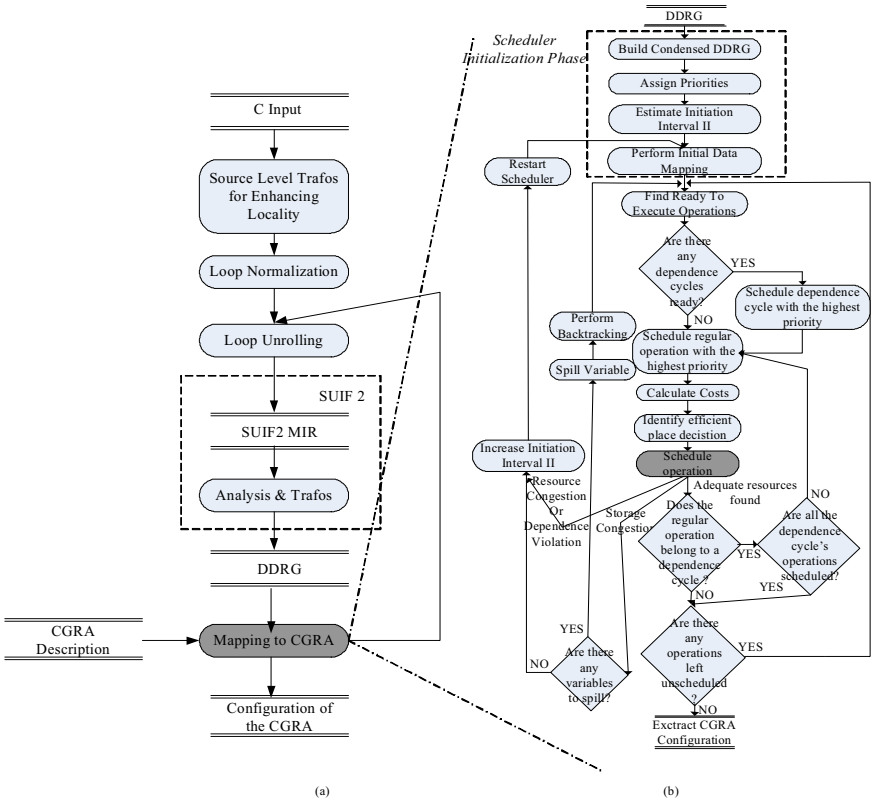

Figure 2. a) Mapping methodology flow for CGRAs and b) Mapping Algorithm

Another important issue is the data allocation strategy followed. More specifically, we perform the scheduling, register allocation and spilling phase in a single step as it was done for the first time in modulo scheduling (for VLIWs processors) in a state of the art technique [20]. It is essential, for having an efficient schedule in a CGRA architecture, the decisions for storage allocation to be performed during the scheduling process. This is due to the fact that the place where the operations' execution takes place as well as the availability of storage resources at a certain PE cannot be estimated before scheduling because different scheduling decisions produce a different data mapping. Additionally, as it was showed in [20] from the combination of the scheduling, register allocation and spilling phase into one step, which is done for VLIW processors, very efficient schedules can be derived. Finally, instead of using the module variable expansion technique used in [17] for the register allocation we use hardware support in terms of rotating register files.

Finally, a data structure is employed for realizing the mapping of variables to storage locations. This data structure records the producer and consumer operations for each variable as well as the storage place after a variable production and consumption action respectively. In this way, when a variable has multiple accesses during its lifetime the scheduler can determine the variable's storage location in the time intervals between its consecutive accesses. Thus it is possible to spill the use of a variable [20] instead of spilling the variable itself. The latter approach requires that a load operation is necessary for all variable's uses when it is spilled. In the first case however only the necessary uses are loaded while the others can be satisfied from the DFM. In architectures with a distributed register file system such as CGRAs this feature gives a large flexibility to the scheduler. Apart from the flexibility, as it is shown in [20], this technique can significantly reduce the unnecessary memory accesses that are generated from spilling.

4.3. Proposed Mapping Algorithm Description

The algorithm firstly identifies the *dependence cycles* and it condenses them to a single node building the condensed DDRG which is acyclic. The operations in this graph are considered ready to execute when all DDPs with zero dependence distance have finished their execution. Next, the priorities of the operations in the condensed

DDRG are estimated. Two types of priorities are assigned to each operation. These are: 1) the mobility [21] and 2) the height [21]. The two priorities ensure that operations residing in the critical path are placed higher in the *ready to execute* operations' priority list. Hence, a higher possibility of faster and more efficient execution in terms of resource reservations is achieved.

The initiation interval is calculated next, as $II = \max(II_{dep}, II_{rec})$ (1) [19], where II_{dep} is the initiation interval imposed by the dependence constraints while II_{rec} is the initiation interval imposed by the resource constraints. Afterwards, the data mapping is initialized. At this point the algorithm places the live-in and live-out variables in the scratch-pad memory while the computations' intermediate variables are assumed to be stored at the PE where they will be generated. In the following it is also assumed that a variable which exists in the CGRA it exists also in the scratch pad so as to satisfy the property of a memory hierarchy.

After the scheduler's initialization phase (Fig.2b), the mapping algorithm schedules the operations one by one, scheduling each time, from the *ready to execute* operations the one which has the highest priority. The scheduler shows preference to the dependence cycles when they are ready since they are more demanding in terms of constraints [17] in respect to single operations. However, for operations with the same granularity (e.g. both single operations) the priorities as described previously are used for defining the sequence of instructions for scheduling.

The aim of the proposed mapping algorithm is to find a cost-effective place and time slot for all operations of the scheduled application. The PE selection for executing an operation, and the way the input operands are fetched to the specific PE will be referred to hereafter as a *Place Decision* (PD) for that specific operation. Each PD has a different impact on the operation's execution time and the way this operation's execution influences the effectiveness of PDs of future scheduled operations. The operation's execution time is determined from the operation's latency, the path delay which is necessary to fetch the operation's operands and the availability of resources. Therefore, large path delay or lack of free resources causes the operation's execution interval to be inflated. Furthermore, larger execution time requires more resources to be reserved for scheduling an operation. Hence, PDs which wastefully consume the CGRA resources cause future schedule instructions to have less cost-effective PD. So, a set of costs, which is described in section 4.3.1, is assigned to each PD to incorporate the aforementioned factors that influence the scheduling of the operations. The algorithm for each operation calculates the costs and examines its schedulability for a possible execution to all CGRA's PEs and chooses the most efficient PD (see section 4.3.1).

The scheduling of an operation in a specific PE finishes normally if the required resources exist. Depending on the availability of resources different actions are performed by the scheduler. In case where the register file size inside a PE is not adequate for finding a valid execution time slot for an operation in the CGRA the algorithm spills the appropriate variables for scheduling the operation. If the variable corresponds to a live-in variable or an intermediate variable already spilled then this variable is overwritten and spilling is not required. If the variable corresponds to an intermediate variable which haven't yet been spilled then a store operation is required. Thus, variables generated from operations can be spilled at most once while live-in variables which exist in the CGRA can be overwritten by other variables and loaded from the scratch-pad when necessary.

Also, for the spilling, the algorithm builds a list of candidate variables which are alive during the time interval of a register conflict at a certain PE. These variables are ordered according to their demands in registers during that interval. Then the algorithm spills one variable at a time until a valid schedule is reached. For each spilled variable the algorithm backtracks to the operation to which the variable belongs in order to introduce the necessary store operation and continues the scheduling process. During the backtracking step, the operations which are dependent on the operation for which the store is introduced are removed from the schedule and put in the unscheduled operations' list with the same priority. Finally, if there are no variables left to be spilled, the algorithm fails for the current initiation interval and the scheduling phase restarts with an increased value of the initiation interval by one.

Additionally, in case where some other resource is not adequate for finding a feasible execution time slot or in case where dependences are violated, the mapping algorithm increases the initiation interval by one and restarts the scheduling process. Finally, the algorithm finishes and produces the CGRA configuration when all operations are scheduled.

4.3.1. Mapping Costs

For finding an efficient PD for each operation, a set of costs was employed. This set of costs is calculated for a possible execution of the operation in each PE in the CGRA. The first one, called *delay cost*, refers to the operation's earliest possible schedule time if it is placed for execution to a certain PE. As shown in eq. (3), it is the sum of the *RTime* plus the maximum of the times *tf* required to fetch the operation's (*Op*) input operands to a specific PE_x, where *P* is the set with the operation's input operands. The $RTime_{Op}$ equals the maximum of the times where each of the *Op*'s DDPs with zero dependence distance (DDP_{Op}) finished executing (tf_{Opi}) (eq.2).

$$RTime_{Op} = \max_{i=1,\ldots,|DDP_{Op}|}\left(tf_{Op_i}, 0\right) \text{ where } Op_i \in DDP_{Op} \quad (2)$$

$$Delay\ Cost\left(PE_x, Op\right) = RTime_{Op} + \max_{i=1,\ldots,|P|}\left(tf_{P[i]}, 0\right) \quad (3)$$

When the operands come from memory, then *tf* equals the memory latency while when they come from a PE in the CGRA equals the time *tr* of routing them to PE$_x$. Hence, by denoting with $PE_{P[i]}$ the PE where the routed operand *P[i]* resides, by t_{init_route} the time at which the routing initiates and by t_{init_fetch} the clock cycle that follows *RTime* where the bus is available to fetch the requested data, we have

$$tf_{P[i]} = \begin{cases} t_{init\_fetch} + memory\ latency, & P[i] \in P_m \\ tr_{PE_{P[i]} \to PE_x, t_{init\_route}}, & P[i] \in P_r \end{cases} \quad (4)$$

where P_m is the subset of P with the operands which are fetched from memory, while P_r=P-P_m, is the subset of P with the operands which are routed to the destination PE*x*.As shown in eq.(4) the time *tr* depends on the t_{init_route} since the availability of the interconnections and storage locations needed for routing an operand depends on the clock cycle where routing initiates.

Furthermore, in this set of costs the *interconnection cost* is also included. It refers to the interconnection resources that need to be reserved for scheduling an operation in a specific PE. When an input operand is routed, the interconnection overhead refers to the interconnections that must be reserved, in order to route the operands to the destination PE. Higher interconnection overhead causes future scheduled operations to have a higher possibility to conflict. As shown in eq. (5), the interconnection cost for a PD is the sum of the CGRA interconnections which are used for routing the operation's input operands.

$$Interconnection\ Cost(PE_x, Op) = \begin{cases} 0, & P_r = \varnothing \\ \sum_{i=1,\ldots,|P_r|} PathLength\left(PE_{P_r[i]} \to PE_x\right) & P_r \neq \varnothing \end{cases} \quad (5)$$

A greedy approach was adopted for calculating the time *tr* (eq.(4)) and the number of interconnections (eq.(5)) required for routing an operand. For each operand the shortest paths, which connect the source and destination PE, are identified. From this set of paths, the one with the minimum routing delay is selected. The length and delay of the selected path gives the delay and interconnection costs through eq.(3) and eq.(5), respectively.

Moreover, a new cost is introduced to address the following problem: Independent operations whose results are used directly or indirectly as input operands to an operation should be placed spatially close in the CGRA. This is depicted in Fig.3a. Although operations A and B are independent to each other, their results are used as direct inputs for the operation C. Hence, they should be placed spatially close. However, as it is shown in Fig.3b operations' A,B,C,D,E and F results would also be used indirectly to produce the input operands of operation L. If the scheduler places the operations as close as possible in the end many parallel operations would execute sequentially due to resource conflicts. For this reason a well balanced approach is required to address this problem.

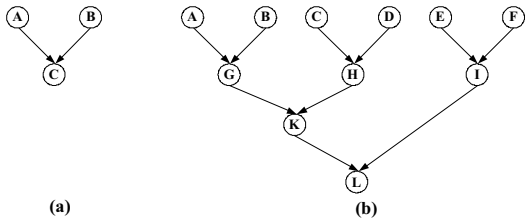

Figure 3. Two sample graphs illustrating the trade-off addressed by the relativity cost.

We have addressed this trade-off by a new cost named *relativity cost*. In order to calculate this cost we analyze the graph for finding independent node pairs whose results are used indirectly or directly to initiate the execution of other descendant operations. We record for each operation of the independent operation pairs its height from the closest common descendant operation. The sum of heights of the pair's nodes equals the relativity cost. For example in Fig.3b the common descendant operation for the two independent operations A and B is operation G. This operation pair has relativity cost equal 2. In the end, the scheduler attempts to place the pair's nodes as close as possible if the value of the relativity cost is below a certain threshold. In our experiments, the value 2 was employed as a threshold as it was proven that it gives the best results in terms of ILP.

In addition, we have introduced the *PE utilization factor* which is calculated from eq.6 and is defined as the ratio of the cycles where a PE is occupied divided by the initiation interval. This heuristic helps operations to spread into the CRA avoiding a possible resource congestion.

$$PE\ Utilization\ Factor = \frac{PE\ Occupation\ cycles}{II} \quad (6)$$

Also, as already described in our previous work [9] there are two ways of accessing a variable that is present both in the CGRA and the scratch pad memory. We follow the procedure described in [9] two identify which of the two ways is the most beneficial. When the way of accessing the data reused values is determined the selected PD for executing the operation is selected as follows: The adopted PD for each operation is the one with the minimum delay cost. In case where there are identical PDs in respect to the delay cost the one with the minimum interconnection cost is adopted. Also, if there are identical PDs in respect to these two costs the PD with the minimum relative cost is adopted. Finally, if there are identical PDs in respect to the aforementioned three costs the PD with the minimum value of the PE Utilization Factor is chosen.

5. Experimental Results

5.1. Experimental Setup

In this section, we present the experimental results from applying the proposed mapping methodology steps on a representative CGRA architecture. We have developed in C++ a prototype compiler framework and a simulation environment for verifying our scheduler operation and performing experiments. The experimental setup considers a CGRA of 16 PEs connected in a 4x4 array. In the experiments two scenarios are considered concerning the PEs' interconnection topologies. The first one (A1) refers to the case where PEs are directly connected to all other PEs in the same row and same column, as in a quadrant of Morphosys [10] (Fig.1b). The second one (A2) refers to the case where each PE is connected only to its nearest neighbours [6] (Fig.1b). The A1 has more available internal bandwidth than A2 due to its richer interconnection topology.

Table 2. Application's characteristics

Application	# of Operations	Unroll Factor	Iterations	Description
fcpx	8	4	200	Fir for complex numbers [22]
Conv	20	1	256	Convolutional filter 3x3 [22]
matmul	29	2	216	matrix multiplication [22]
fft	80	1	16	radix-4 FFT [22]
iir	39	1	100	iir filter [22]
wave_ver	64	1	16	vertical pass (2D-DWT) [22]
wave_hor	18	1	128	horizontal pass (2D-DWT) [22]
latanal	18	2	100	lattice analysis filter [22]
latsynth	18	2	100	lattice synthesis filter [22]
volterra	27	1	100	iir filter
nc	56	1	100	noise cancelling
wdf	44	1	100	iir filter
fdct_ver	60	1	64	vertical pass (2D-FDCT) [22]
fdct_hor	61	1	64	horizontal pass (2D-FDCT) [22]
idct_ver	61	1	64	vertical pass (2D-IDCT) [22]
idct_hor	79	1	64	horizontal pass (2D-IDCT) [22]

Additionally, each PE has a register file of size 16 words with two input ports and four output ports. There is one FU in each PE that can execute any operation in one clock cycle. The granularity of the FU is 16-bit, which is the word size. The direct connection delay among the PEs is zero cycles. Furthermore, two buses per row are dedicated for transferring data to the PEs from the scratch-pad memory. Each bus transfers one word per scratch pad's memory cycle. Additionally, we assume that the CGRA's Context Caches have size of 16 context words. Finally, in order to delineate the impact of the memory access latency to the performance and operation parallelism we assume for our measurements that the memories access latencies are constant for each scenario.

We have used 16 characteristic DSP applications written in C code. The first set consists of 13 programs drawn from the Texas Instruments DSP benchmark suite [22]. Their characteristics are given in Table 2. More specifically, the second column refers to the number of operations in the application's loop body, the third one refers to the times the applications' loops have been unrolled, the fourth one refers to the number of iterations of the applications' loops, while the fifth one contains a brief description for each application.

5.2. Experimentation

5.2.1. Performance Improvements

Fig.4 and 5 show the performance comparison for mapping the designs on the CGRA, with and without exploiting data reuse opportunities. We consider 4 scenarios concerning the memory access latency. Also, the measurements in Fig.4 correspond to the A1 architecture, while Fig.5 refers to the A2 case. Above the bars, the percentages of performance improvements are shown.

In the A1 case for memory latency 1 cycle half of the algorithms run faster when data reuse opportunities are exploited. In that case, the improvements range from 0% to 49%. For larger values of the memory latency the improvements become larger. On average the performance is improved by 37,1% if we consider all scenarios for the value of the memory latency. For the A2 architecture alternative, smaller improvements were achieved in comparison to the A1 case. For memory latency 1 cycle in the A2 architecture, the performance improvements range from 0% to 32,6%, while, performance is improved by 32,6% on average in the A2 case.

In [19], it was stated that the demand each algorithm has in respect to the CGRA's resources determines the initiation interval *II* from which the performance depends. The CGRA resources that are exhaustively consumed during the execution determine the value of the initiation interval. We will call them to hereafter as *critical resources* for better clarification. In our case, the algorithms having the bus as critical resource and plenty of data reuse opportunities showed significant improvements even for 1 cycle access latency. Moreover, algorithms which also have the CGRA interconnection network as a critical resource showed higher improvements in the A1 architecture than in A2 and no improvement for memory access latency 1 cycle.

Also, Fig.4 and 5 show that small performance improvements were derived for memory access latency 1 cycle. Moreover, these improvements increase as the memory latency increases. This is due to the fact that for memory latency 1 clock cycle it is not always beneficial to route the data reuse values through the internal interconnection network. Moreover, this becomes more obvious in the A2 case where due to the poorer PE connectivity smaller improvements are achieved. In our previous work [9] we have addressed this problem by introducing a threshold (*memory_thresh*) for deciding whether it is beneficial to route a data reuse value or fetching it from memory. In this work we have used the optimized value of this threshold for each application to derive our measurements.

Fig.6a, shows the average IPC for all benchmarks with and without data reuse exploitation for the two architecture alternatives (A1 and A2) while in Fig.6b the improvements of the average IPC are illustrated in respect to the memory access latency. It is deduced that for memory latency larger than 3 cycles the improvements for the two architectures equalize. More specifically, the

improvements are 90% on the average. This happens because the initiation interval is increased (due to the increased memory access delay) to the point where the interconnection network is not critical anymore. For smaller values of the memory latency the CGRA interconnection network becomes important especially for the value of 2 cycles.

5.2.2. Storage Requirements

Fig.7 illustrates the impact of the register file size on the average value of IPC, for all benchmarks, with and without the data reuse exploitation for the A1 and A2 architectures. As illustrated, a small register file has higher impact on the IPC as the memory latency increases. This is because the higher value of memory latency increases the demands in storage locations. Moreover, the average value of IPC for small values of the register file size in the data reuse case, drops faster. This is explained as follows:

When data reuse opportunities are exploited more data values are stored in the DFM and this increases the possibility of a storage conflict. Moreover, the spilling of variables that inevitably happens, burdens the buses with additional memory accesses and this reduces the ILP due to bus conflicts.

Also, for small register files sizes the data reuse exploitation case tend to behave similarly in respect to performance with the case where data reuse opportunities are not exploited. This is expected since the optimization performed by the application of our methodology is based on the ability of the DFM to store and route data reused values. However, as it is shown even with a small register file significant improvements can be achieved.

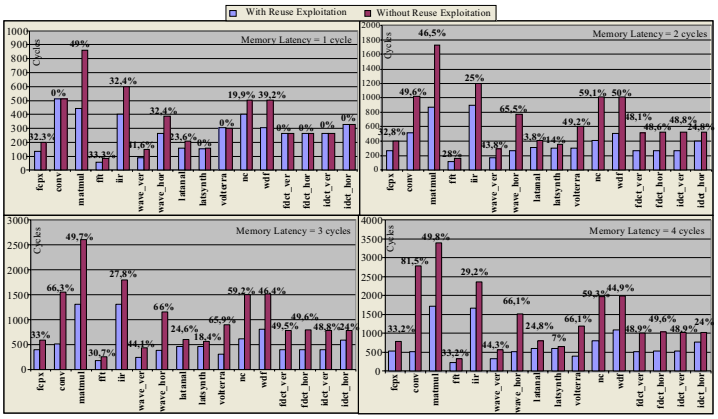

Figure 4. Performance comparison with and without data reuse exploitation (A1).

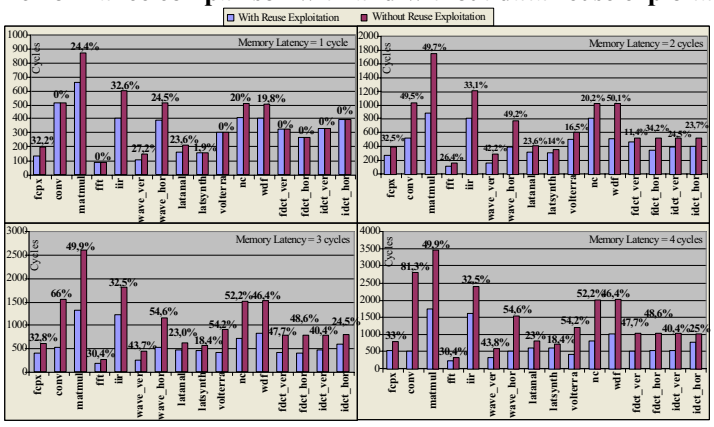

Figure 5. Performance comparison with and without data reuse exploitation (A2).

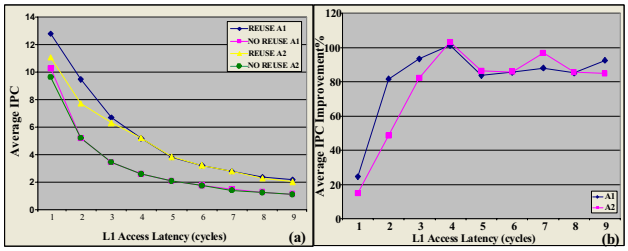

Figure 6. a) Average IPC and b) average IPC improvements% in respect to memory access latency

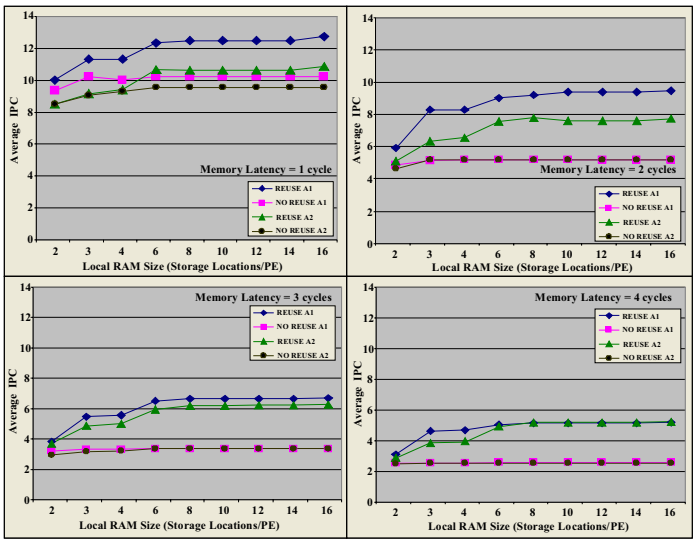

Figure 7. Average IPC in respect to Local RAM Size

6. Conclusions

In this work an optimized mapping approach for mapping applications to CGRAs was presented. A set of heuristics was introduced for efficient mapping taking into account the routing of data values through the interconnection network. Finally, the parametric CGRA architecture template was exploited so as to explore the design space formed by the proposed methodology.

7. Acknowledgements

We thank the European Social Fund (ESF), Operational Program for Educational and Vocational Training II (EPEAEK II), and particularly the Program HERAKLEITOS, for funding the above work[1].

References

[1] R. Hartenstein, "A decade of reconfigurable computing: A visionary retrospective",*Proc. of ACM/IEEE DATE '01*, pp. 642-649.

[2] Pact Corporation, "*The XPP white Paper*", Technical report, www.pactcorp.com, 2005.

[3] B. Mei, S. Vernalde, D. Verkest, R. Lauwereins, "Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture, A Case Study", in *Proc. of DATE '04*, pp. 1224-1229.

[4] E. Waingold, M. Taylor, D. Srikrishna, *et.al.,* "Baring it all to Software: Raw Machines", in IEEE Computer, vol. 30, no. 9, Sept 97, pp 86-93

[5] F. Catthoor, K. Danckaert, C. Kulkarni, *et.al.*, "*Data Accesses and Storage Management for Embedded Programmable Processors*", Kluwer Academic Publishers, 2002.

[6] Reiner W. Hartenstein and Rainer Kress, "A Datapath Synthesis System for the reconfigurable datapath architecture", in Proc. of ASP-DAC, Article No.77, Sep. 1995

[7] J.M.P Cardoso and M. Weinhardt, "XPP-VC: A Compiler with temporal partitioning for the PACT-XPP architecture" in *Proc. of* FPL 02, LNCS 2438, Springer-Verlag, pp. 864-874, 2002

[8] J. Lee, K. Choi and Nikil Dutt, "Compilation Approach for Coarse-grained Reconfigurable Architectures", in IEEE Design & Test of Computers, vol. 20, no. 1, pp. 26-33, Jan.-Feb., 2003.

[9] G. Dimitroulakos, M.D Galanis, C.E. Goutis, "Alleviating the Data Memory Bottleneck in coarse grained reconfigurable arrays", Proc. IEEE ASAP Conf. July 2005 pp 161-168.

[10] H. Singh, L. Ming-Hau, L. Guangming, *et.al.*, "MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Communication-Intensive Applications", in IEEE Trans. on Computers, vol. 49, no. 5, pp. 465-481, May 2000.

[11] Scott A. Mahlke, David C. Lin, William Y. Chen, *et.al.*,"Effective Compiler Support for Predicated Execution Using the Hyperblock.", Proc. 25th Microarchitecture,pp 45-54, 1992.

[12] K. Kennedy and R. Allen, "*Optimizing Compilers for modern architectures*", Morgan Kauffman Publishers, 2002.

[13] B.R. Rau, M. Lee, P. Tirumalai and M.S. Schlansker, "Register Allocation for Software Pipelined Loops", in Proc. of ACM SIGPLAN PLDI, pp. 283-299, June 1992

[14]P. R. Panda, N. Dutt, and A. Nicolau, "*Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*", Kluwer Academic Publishers, 1999.

[15] P. R. Panda, F. Catthoor, N. D. Dutt, *et.al.*, "Data and Memory Optimization Techniques for Embedded Systems", in ACM Trans. on Design Automation of Electronic Systems (TODAES), vol. 6, no.2, pp. 149-206, April 2001.

[16] M. W. Hall et al., "Maximizing multiprocessor performance with the SUIF compiler", Computer, vol. 29, pp. 84-89, 1996.

[17]M.S.Lam, "Software pipelining: An effective scheduling technique for VLIW machines",*Proc. of SIGPLAN '88*, pp. 318-328.

[18] B.R. Rau, "Some Scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing", Proc. 14th Ann. Microprogramming Workshop, pp. 183-197, Oct. 1981.

[19] B.R. Rau, "Iterative Modulo Scheduling: An algorithm for software pipelining loops", Proc. 27th Ann. Int'l Symp. Microarchitecture, pp. 63-74, San Jose, Calif.,Dec. 1994.

[20] Javier Zalamea, Josep Llosa, Eduard Ayguade and Mateo Valero, "*Register Constrained Modulo Scheduling*", in IEEE Trans. on Par. and Distr. Syst., Vol 15, No 5, May 2004, pp 417-430.

[21] G. De Micheli, "*Synthesis and Optimization of Digital Circuits*", McGraw-Hill, International Editions, 1994.

[22] Texas Instruments Inc., www.ti.com, 2005.

1