Optimized Mapping for enchancing the operation parallelism in Coarse-Grained Reconfigurable Arrays

GREGORY DIMITROULAKOS [1], MICHALIS D. GALANIS[2], COSTAS E. GOUTIS[3]
VLSI Design Laboratory, Electrical & Computer Eng. Dept., University of Patras, Greece

Abstract: It is widely known that bandwidth limitations degrade parallel systems' performance. This paper presents a mapping methodology for coarse-grain reconfigurable arrays which alleviates the bandwidth bottleneck by exploiting the processing elements interconnection network for transferring values with data reuse opportunities. A novel mapping algorithm is also proposed that uses a resource-aware modulo scheduling technique. From the application of the proposed mapping approach, significant improvements in performance were achieved while we have also quantified these improvements in respect to crucial architecture parameters such as the memory latency and the register file size. For this reason, our methodology targets on a parametric architecture template which can model a large number of existing architectures of this kind.

Key-Words: - Reconfigurable embedded systems, Coarse-grain reconfigurable array, operation parallelism, mapping, scheduling, data bandwidth optimization.

1. Introduction

Coarse-grain reconfigurable architectures have been proposed for accelerating loops in several scientific domains' in embedded systems. These architectures combine the high performance of ASICs with the flexibility of microprocessors. Coarse-grain reconfigurable architectures consist of a large number of Processing Elements (PEs) connected with a configurable interconnect network. This work focuses on architectures where the PEs are organized in a 2-Dimensional (2D) array and they are connected with mesh-like reconfigurable networks [1] and [2]. In this paper, these architectures are called Coarse-Grain Reconfigurable Arrays (CGRAs). 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 to the mesh-like interconnect. Also, their coarse granularity greatly reduces the delay, power and configuration time relative to an FPGA device at the expense of flexibility.

It is widely known that parallel operation execution in parallel systems generates a respective increase in memory accesses. Since the memory interface provides a limited access bandwidth, the applications performance cannot be that high as the parallel system capabilities promise. This problem is known as the *memory bandwidth bottleneck* [3] and restraints the exploitation of the inherent parallelism. Thus, a mapping methodology to CGRAs for reducing the memory bandwidth is required.

This paper presents a memory-aware mapping methodology for CGRAs that attempts to minimize the data memory bandwidth requirements by exploiting applications' data reuse opportunities and the architecture's foreground memory. The high bandwidth Distributed Foreground Memory (DFM) which is constituted from the PEs' register files and the interconnections among them is exploited for the purpose of relieving the external memories from the data transfer burden. In this way more operations which require memory accesses can be run in parallel. Additionally, accessing data from the foreground memory is generally faster than accessing data from the external memory. Hence, by increasing parallelism and reducing the average memory access time, the performance is increased. A novel mapping algorithm is also proposed that uses a modulo scheduling technique in which the binding, routing, and scheduling phases are considered together and they are steered by a set of costs. The experimental results showed that performance can be improved a lot from our mapping approach. Moreover, the experimental results quantified the impact of the architecture's parameters (memory latency and register file size) on performance and Instructions Per Cycle (IPC) for a representative set of DSP applications.

The rest of the paper is organized as follows: section II describes the related work, while section III presents the considered architecture template. Section IV describes the proposed mapping methodology while Section V our mapping algorithm. Section VI describes the mapping costs. The experimental results are presented in section VII. Finally, conclusions are outlined in section VIII.

2. Related Work

Although several CGRA architectures have been proposed in the past few years [1]-[2], only a few methodologies ([4]-[7]) have been proposed for tackling the memory bandwidth bottleneck. None of these have illustrated how the architecture parameters affect the performance improvements. Furthermore, they haven't considered the case where the foreground storage size is limited. Our methodology encounters as well the case where the foreground storage size is limited using variable spilling.

In [4] a CGRA architecture was presented. For reducing the average memory access time, the mapping methodology uses a global register file for storing frequently reused data values. However, the single bus (rDPA bus) which transfers data from the global register file to the PEs does not increase the available bandwidth as it is the case when the DFM is exploited.

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 [5] 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 [6] 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 [7] we had proposed a list scheduling technique to reduce the data transfer bottleneck by using the DFM. The current work achieves better improvements with the proposed modulo scheduling technique. Hence, we consider our current work as an enhanced version of our previous work.

3. Generic Architecture Template

In this section, the generic reconfigurable architecture template to which our methodology targets is presented. It is parametric and it is based on characteristics found in the majority of CGRAs ([1] and [2]) so as for our methodology to be retargetable and representative for most CGRA architectures. It consists of 4 basic parts (Fig.1): the PEs organized in a 2D interconnect network, the configuration memory and the memory interface which includes the buses, scratch-pad memory and the main data memory.

In this template, each PE contains one Functional Unit (FU), which it can be configured to perform word-level operations, identical to the ones supported by the operators of the C language (ALU, shifts e.t.c.). For storing intermediate values and values fetched from memory, a register file exists inside a PE. Each register file is a rotating one for realizing the register renaming mechanism which is necessary for modulo scheduling. Moreover, each FU accepts input data that can come from three different sources: a) from the same PE, b) from another PE and c) from the memory buses. The output of each FU can be routed to other PEs through the register file. Also, a context cache inside each PE stores context words that determine, for each PE, how the FU, the storage unit and the communication with neighbouring PEs is configured.

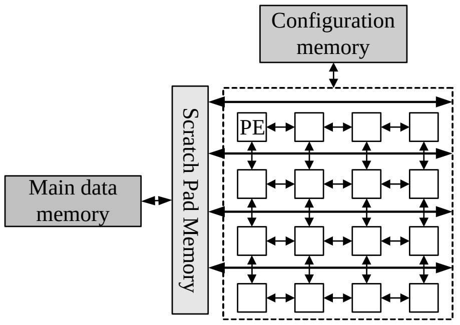

Figure 1. CGRA architecture template

The CGRA's memory interface consists of a scratch-pad memory , the memory buses and the main memory module as shown in Fig. 1. The PEs residing in a row, share a common bus connection to the scratch-pad memory (Fig.1). This also happens in popular CGRA architectures like [8]. The scratch-pad memory is located between the array of the PEs and the main memory, and provides the array with the required data bandwidth. Finally, the configuration memory of the CGRA stores the whole configuration for setting up the CGRA for executing the application's loops. The context caches inside PEs are used for the fast reconfiguration of the CGRA.

4. Mapping Methodology

Fig.2 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 [3]. In this way, for a given size of DFM more data reused values can exploit it instead of using the buses. Afterwards, the loop normalization transformation [9] 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 [10]. 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.2.

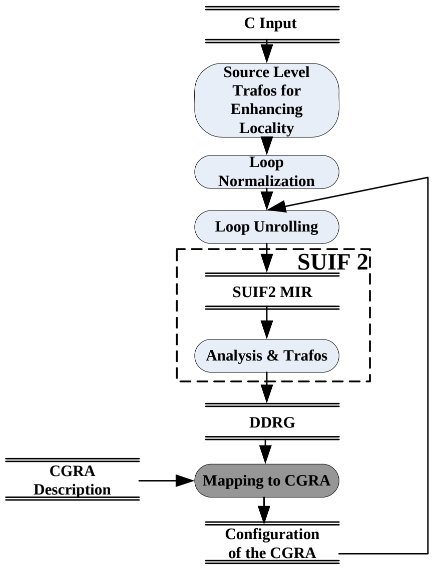

Figure 2. Mapping Methodology

Finally**,** the DDG of the loop body produced by the SUIF compiler is the first input to the mapping algorithm, with the extra information concerning the data reuse among operations. We call this graph *Data Dependence Reuse Graph* (DDRG). The second input to the mapping phase is the CGRA description which is described in terms of a graph, called CGRA Graph. By taking these two inputs, the modulo scheduling algorithm produces the configuration to the CGRA.

4.1 **M**apping **A**lgorithm

The proposed modulo scheduling algorithm is based on the two stage hierarchical reduction technique described in [11], which can be applied to VLIW processors. This approach was changed properly so as to be applied efficiently in a CGRA architecture. Moreover, the scheduling, register allocation and spilling phase are performed in a single step as it was done for the first time in modulo scheduling (for VLIWs processors) in [12]. Additionally, for the variable spilling a similar approach as in [12] was followed.

As shown in Fig.3, the first input to the mapping algorithm is the DDRG. The DDRG is generally a cyclic directed graph $G(V, E, E_\mathrm{R})$, 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 the operations. Each dependence edge E is annotated with the type of dependence as well as with the dependence distance [9]. 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. The *data reuse dependence distance* equals the number of iterations between subsequent uses of the common variable.

The CGRA graph is the second input to the mapping phase. The CGRA graph is an undirected graph, $G_\mathrm{A}(\ V, E_I\ )$ where V is the set of CGRA's PEs and E_I the interconnections among them. The CGRA's description includes also parameters, like the PEs' register file size, the memory buses to which each PE is connected, the bus bandwidth, the scratch-pad's memory access times and the CGRA's resource reservation record. The latter one records the reservations performed and has the structure of a modulo reservation table [13].

The algorithm firstly identifies the *dependence cycles* and it condenses them to a single node building the condensed DDRG which is acyclic. Next, the priorities of the operations in the condensed DDRG are estimated. The priority of an operation equals its *height* [14]. However in case where two operations have the same *height* the operation with the smallest value of *mobility* [14] is considered first in the scheduling phase. Afterwards, the initiation interval is calculated as $II = \max(II_{dep}, II_{rec})$ [13], where II_{dep} is the initiation interval imposed by the dependence constraints while II_{rec} is the initiation interval imposed by the resource constraints. Subsequently, 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.

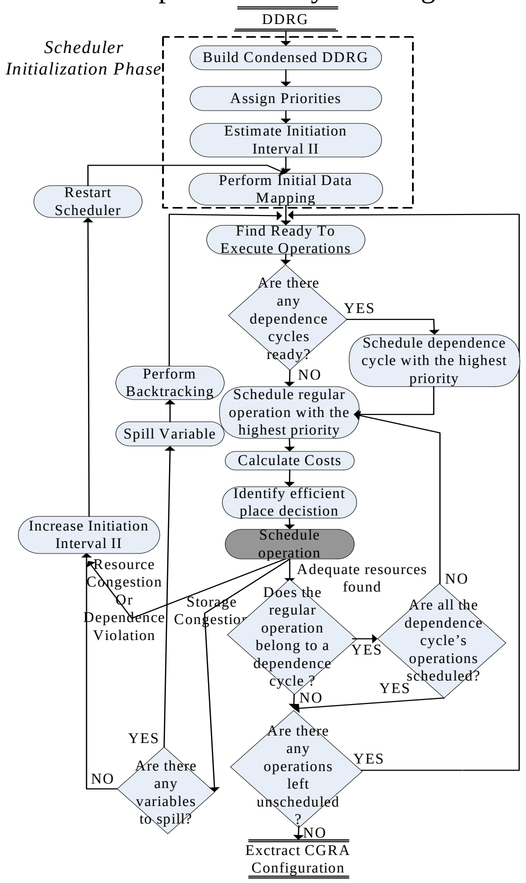
DDRG
Scheduler Initialization Phase
Build Condensed DDRG
Assign Priorities
Estimate Initiation Interval II
Perform Initial Data Mapping
Restart Scheduler
Find Ready To Execute Operations
Are there any dependence cycles ready?
YES
Schedule dependence cycle with the highest priority
NO
Perform Backtracking
Schedule regular operation with the highest priority
Spill Variable
Calculate Costs
Identify efficient place decistion
Increase Initiation Interval II
Schedule operation
Adequate resources found
NO
Resource Congestion Or Dependence Violation
Storage Congestion
Does the regular operation belong to a dependence cycle ?
YES
Are all the dependence cycle's operations scheduled?
YES
NO
YES
Are there any operations left unscheduled ?
YES
NO
Are there any variables to spill?
NO
Exctract CGRA Configuration

Figure 3. Mapping Algorithm

After the scheduler's initialization phase (Fig.3), the mapping algorithm iterates for scheduling all operations one by one, scheduling each time, from the *ready to execute* operations the one which has the highest priority. The operations are considered ready to execute when all DDPs with zero dependence distance are scheduled. The *dependence cycles* have higher priority from the single operations since they require more resources to be scheduled and the initiation interval is heavily influenced by them.

Afterwards, the PE where the operation will be executed is determined. 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. A set of costs which is described in section VI, is used for identifying an efficient PD for executing each operation.

In the next step, the actual scheduling of the operation takes place. The scheduling of an operation finishes normally if its execution satisfies the data dependences and if there is no resource conflict with the already scheduled operations. Depending on the availability of resources different actions are performed by the scheduler (Fig.3). In case where the register file size inside the PEs 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. The algorithm spills the appropriate variables using the heuristics applied in [12]. Then, the algorithm backtracks to the operation which fetches the variable for introducing the necessary store operation and resumes the scheduling process. 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 conflict occurs in respect to some other resource or in case where dependences are violated, the mapping algorithm increases the initiation interval by one and restarts the scheduling process.

4.2 **Mapping Costs**

For finding an efficient PD for each operation, a set of costs was employed. 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. The value for each cost depends on the PE where the operation is executed. 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. This cost depends both from the routing delay as well as from the availability of resources. The second is called *interconnection cost* and equals the number of interconnections utilized for transferring the data reuse values. The third one is called *memory cost* and equals the number of memory accesses required for each operation. Finally, we have introduced the *PE utilization factor* which is defined as the ratio of the cycles where a PE is occupied divided by the initiation interval. In our previous work [7] we describe in detail how the costs are calculated. Moreover, 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 [7] to 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 the one with the minimum delay cost. If there are multiple PDs with the same delay cost the one that minimizes the memory cost is selected. In case where there are identical PDs in respect to the two aforementioned costs the one with the minimum interconnection cost is adopted. Finally, if there are identical PDs in respect to these three costs the one with the minimum value of *PE Utilization Factor* is chosen.

5 **Experimental Results**

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 that realizes our mapping algorithm. The experimental setup considers a 2D CGRA of 16 PEs connected in a 4x4 array. The PEs are directly connected to all other PEs in the same row and same column, as in a quadrant of Morphosys [8]. Each PE is assumed to have a register file of size 16 words. There is one FU in each PE that can execute any operation in one clock cycle. The interconnect delay on direct connection among the PEs is 0 cycles. Also, two buses per row are dedicated for transferring data to the PEs from the scratch-pad memory while, each bus can transfer one word per cycle. Additionally, we assume that the CGRA's context caches have size of 16 context words.

Table 1. Benchmarks' characteristics

Application	# of Operations	Unroll Factor	Iterations	Description
fcpx	8	4	200	Fir for complex numbers
matmul	29	2	216	matrix multiplication
fft	80	1	16	radix-4 FFT
iir	39	1	100	iir filter
wave_ver	64	1	16	vertical pass (2D-DWT)
wave_hor	18	1	128	horizontal pass (2D-DWT)
latanal	18	2	100	lattice analysis filter

We have used 7 DSP applications taken from the TI benchmark suite [15]. Their characteristics are given in Table 1. More specifically, the second column gives the number of operations in the application's loop body, the third one refers to the times that 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. These algorithms are characterized by high data transfer rate between the CGRA and the memory. Moreover, a considerable amount of data reuse opportunities exists. Thus, they can be considered as representative testbench for evaluating our approach. 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.

From Fig.4 it is deduced that considerable improvements were achieved by the application of our scheduling technique. On average, performance is improved by 38% if we consider all scenarios of the memory access latency. Additionally, for the considered set of benchmarks the improvements have proven to be independent from the memory access latency. Hence, applications which are characterized by high data transfer rate and a respective amount of data reuse opportunities exhibit the same behaviour in performance for the different scenarios of the memory access latency. However, as we have experimentally investigated this is not the case for algorithms with other characteristics. In this paper, we have omitted such an exploration due to space limitations.

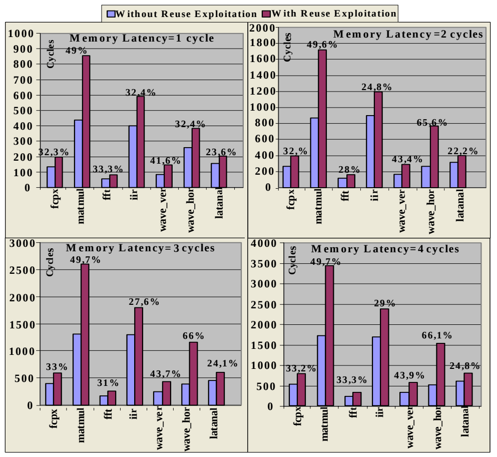

Figure 4. Performance in respect to memory latency

Finally, Fig.5 illustrates the impact of the register file size on the average value of IPC with and without exploiting the data reuse opportunities. The average value of IPC for small values of the register file size when data reuse opportunities are exploited drops faster than the case where they are not exploited. 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 congestion* state. The spilling of variables that inevitably happens, burdens the buses with additional memory accesses and this reduces the operation parallelism due to bus conflicts.

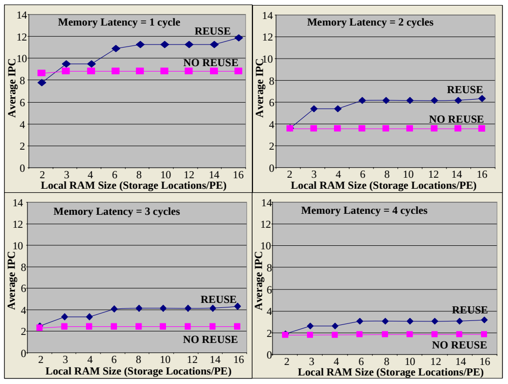

Figure 5. Average IPC in respect to PEs' register file size

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.

6 Conclusions

It is deduced that our mapping approach achieved to improve the performance along with the operation's parallelism by exploiting the data reuse opportunities for reducing the memory fetches. Additionally we have tried to quantify these improvements in respect to important architecture characteristics such as the memory latency, and the register file size.

References

[1] R. Hartenstein, "*A decade of reconfigurable computing: A visionary retrospective*", in *Proc. of ACM/IEEE DATE '01*, pp. 642-649, 2001.

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

[3] F. Catthoor et al., "*Data Accesses and Storage Management for Embedded Programmable Processors*", Kluwer Academic Publishers, 2002.

[4] Reiner W. Hartenstein and Rainer Kress, "*A Datapath Systhesis System for the reconfigurable datapath architecture*", ASP-DAC,Sep 1995

[5] Joao M.P Cardose and Markus Weinhardt,"*XPP-VC: A Compiler with temporal partitioning for the PACT-XPP architecture*",FPL 02

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

[7] 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

[8] H. Singh 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.

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

[10] SUIF2 compiler, http://suif.stanford.edu/suif/suif2/, 2006.

[11] M.S. Lam, "*Software pipelining: An effective scheduling techniquefor VLIW machines*",in Proc of SIGPLAN 88,pp 318-328.

[12] 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

[13] 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.

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

[15] Texas Instruments Inc., www.ti.com, 2006.