Software Pipelining for Coarse-Grained Reconfigurable Instruction Set Processors

Francisco Barat, Murali Jayapala, Pieter Op de Beeck and Geert Deconinck

K.U.Leuven, Belgium.

{f-barat, j4murali}@ieee.org, { Pieter.OpdeBeeck, Geert.Deconinck}@esat.kuleuven.ac.be

Abstract

This paper shows that software pipelining can be an effective technique for code generation for coarse-grained reconfigurable instruction set processors. The paper describes a technique, based on adding an operation assignment phase to software pipelining, that performs reconfigurable instruction generation and instruction scheduling on a combined algorithm. Although typical compilers for reconfigurable processors perform these steps separately, results show that the combination enables a successful usage of the reconfigurable resources. The assignment algorithm is the key for using software pipelining on the reconfigurable processor.

The technique presented is also able to exploit spatial computation inside the reconfigurable functional unit by which the output of a processing element is directly connected to the input of another processing element without the need of an intermediate register. Results show that it is possible to reduce the cycle count by using this spatial computation.

1 Introduction

VLIW cores are nowadays the processor of choice when implementing multimedia applications on low cost terminals [1]. Reconfigurable instruction set processors (RISP) [2] can potentially reduce the power consumption of high performance multimedia applications by fusing the concept of a reconfigurable array with a programmable processor. By using reconfigurable hardware, it is possible to reduce the number of instructions executed when compared to a VLIW processor due to the fact that the instruction set is better adapted to the application. A single reconfigurable instruction accounts for several normal instructions, thus the number of executed instructions decreases (and thus the corresponding power consumption from program memory accesses).

An added benefit is that reconfigurable hardware allows for spatial computation [3], which can be used to improve performance even further. Spatial computation allows outputs from one processing element of the reconfigurable array to be directly connected to the inputs of another processing element. Typically, only temporal computation is used in standard processors (i.e. the output from a processing element is always stored in an intermediate register). Temporal computation limits the performance of the processor by forcing all elements to use the same base clock cycle while spatial computation allows a better division of the time budget.

Code generation for reconfigurable instruction set processors is extremely difficult due to the complexity of the hardware. Not only the code has to be generated but also instructions have to be created and used within the generated code. On multimedia applications, loops are typically the most processor intensive part of multimedia applications. Optimization of inner loops is therefore of special relevance in the multimedia domain.

The aim of this paper is to show that software pipelining [8], a technique typically used to optimize loops for VLIW (very long instruction word) and super scalar processors, can be an effective technique for instruction and code generation of coarse-grained reconfigurable processors, a class of reconfigurable processors of special importance in multimedia applications. In software pipelining, iterations are initiated at regular intervals and execute simultaneously but in different stages of the computation. This allows an increase in the amount of available parallelism that is exploited with the huge number of resources that reconfigurable processors have. Software pipelining has been tested on CRISP [7], which is a reconfigurable instruction set processor template designed for low power multimedia applications.

Section 2 describes previous work. Section 3 describes CRISP, the target processor of the code generation technique and section 4 presents the code generation method. Finally, section 5 presents results followed by conclusions in section 0.

2 Previous work

The majority of current reconfigurable processors are based on fine-grained reconfigurable logic. The coupling

between the reconfigurable logic and the processor determines how the compiler will work.

Compilers for tightly coupled processors usually work in a two-phase mode: first, new instructions are created, and then, they are scheduled with the rest of the code. The most widely used technique [4], analyzes the data dependence graph for operation nodes that can be collapsed onto a single, more complex, node with multiple inputs and a single output (MISO). The MISOs found are then scheduled with the rest of the code as if they were normal operations. The drawback of this approach is that only a connected part of the data dependence graph can be mapped onto the reconfigurable unit. It is not possible to map two or more non-connected graphs at the same time, which thereby limits the obtained performance.

For less tightly couple reconfigurable logic, the usual approach is to map complete loops in the reconfigurable logic, like in [6]. The work in [5] combines the complete loop into a configuration of the reconfigurable processor. The loop is combined into a hyperblock in order to increase the parallelism available. Unfortunately, the method only works if the loop fits in the reconfigurable array.

3 The target processor

CRISP (Configurable and Reconfigurable Instruction Set Processor) is an instruction set processor that is configurable at design time and reconfigurable at run time [7]. It is a VLIW processor composed of a number of fixed functional units (FFUs) and a reconfigurable functional unit (RFU). An example instance can be seen on Figure 1. As all VLIW processors, the processor executes instructions that are composed of parallel operations. Operations are executed in the functional units (both fixed and reconfigurable).

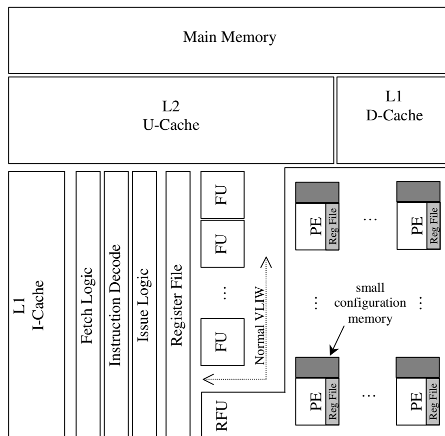

Figure 1 Example CRISP instance

The fixed functional units are the functional units found in typical VLIW processors, such as an integer unit, a multiplier or a load/store unit. The reconfigurable functional unit externally looks like the other functional units but its behavior is reconfigurable. It receives an opcode (operation code), which will determine the operation to be performed, and one or more operands from the register file. As a result, it produces one or more results that are written to the register file. Translation from opcode to the actual control lines driving the hardware is not done in the usual manner. Instead of a hardwired decoder, the reconfigurable unit has a reconfigurable decoder. This reconfigurable decoder is nothing more than a configuration memory addressed by the opcode. The opcode determines which configuration is used in the reconfigurable array. By modifying the contents of the configuration memory, it is possible to change the way in which the reconfigurable unit behaves. The instruction set of the processor is thus modified.

Seen from the inside, the reconfigurable functional unit is a reconfigurable array of coarse-grained processing elements (PEs). Each processing element is a copy of one of the fixed functional units. The usage of such complex units allows for an increase in the performance of the array while also drastically reducing the power consumption when compared to more traditional approaches such as look up tables (LUTs). This is true because the application domain of CRISP (multimedia applications) and the hardware have similar granularities (data sizes).

The processing elements are connected together through a full crossbar (not depicted on Figure 1). This crossbar can connect the output of any processing element to the input of any other processing element. It is also possible to connect a processing element to a register from the main register file through the RFU ports. In the example processor, each cycle two registers can be read from the register file and one register can be written (i.e. two input ports and one output port).

Each processing element has a register at its output (see Figure 1) that can be optionally bypassed, just like in traditional FPGAs. By combining this optional register and the crossbar, it is possible to perform spatial computation. Elements in a data flow chain are connected together through the crossbar. The processing element at the end of the chain is registered to combine temporal and spatial computation.

Only the fixed functional units are needed in order to execute any program. The reconfigurable array is used to increase performance and decrease power consumption.

In the case of CRISP, reconfigurable instructions are basically expanded onto a set of VLIW instructions executed with the processing elements inside the RFU. It is a way to increase the parallelism of the processor without the cost of reading an extremely long instruction from program memory. The instructions are instead read

from the configuration memory, which is a small local memory that has low power consumption. As reloading the configuration memory takes time and power, it is important to minimize reconfiguration times.

4 Loop optimization

This section presents a modified version of software pipelining for coarse-grained reconfigurable instruction set processors that eliminates the drawbacks of the methods introduced in section 2.

The starting point for any software pipelining algorithm is a data dependence graph (DDG) that can express normal and loop carried dependences. Figure 2 is an example DDG that will be used to illustrate different steps in the algorithm. The latency of each arc is the latency from Table 1. The graph is composed of two non-connected sub graphs, one that performs computations and another one in charge of keeping track of the iteration count. In the graph, loop carried dependences are shown with a dashed arrow.

Table 1 Latencies used in the example

	FFU	RFU
LD/ST	2	-
AU	1	0.25
LU	1	0.25
B	1	-
SHIFT	1	0.5

In order to use software pipelining on the reconfigurable processor, it is necessary to realize that the processing elements in the reconfigurable array have a similar (or identical) complexity to the fixed functional units of the processor, and that the processing elements and functional units can all be used simultaneously. With this in mind, the reconfigurable processor can be seen as a traditional VLIW processor with two datapath clusters: a main cluster (MC), composed of the fixed functional units, and a reconfigurable cluster (RC), composed of the processing elements of the reconfigurable array.

Some operations in the reconfigurable processor can be performed in both clusters. An operation assignment phase to decide in which cluster an operation is performed has been added before each scheduling attempt. This assignment phase will place each operation in one of the two clusters.

The modified software pipelining algorithm is as follows (MII, minimum iteration interval and II, iteration interval):

1. Calculate the MII and set II = MII.
2. Perform operation assignment until a valid assignment is found. If unsuccessful, increment II and go to 2.
3. Do modulo scheduling on the loop with II and with the assignment found in step 2. If failure, increment II and go to 2.

The iteration interval (II) is the delay between the initiations of consecutive iterations in a software pipelined loop. It determines the average number of cycles that will be spent per iteration. The MII is the minimum iteration interval that the loop can have. Once the MII is estimated, an attempt to assign and schedule the loop with the II set to MII is made. If no solution is found to either step, the iteration interval is increased by one and a new assignment and scheduling attempts are made. This process continues until a valid assignment and schedule are found or a maximum II is reached. In the latter case, software pipelining cannot be applied to the loop and the loop is generated without software pipelining.

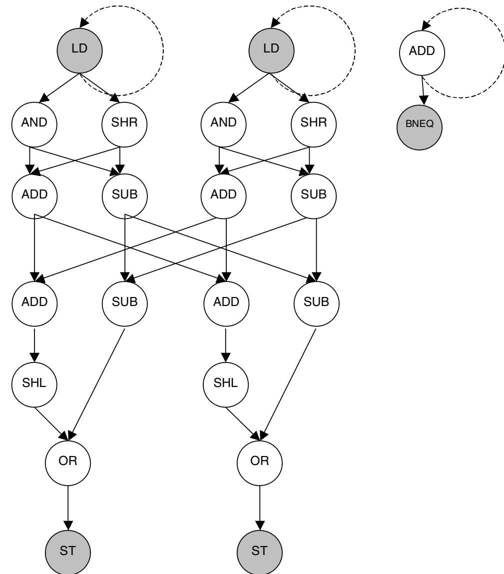

Figure 2 Example data dependence graph. The coloring (see 4.1) represents the graph after initial assignment. Grey nodes are in the main cluster. White nodes are in the reconfigurable cluster.

The MII is due to resource and dependence constraints. The final MII is the maximum of the MII imposed by each type of constraint. For an explanation on how to calculate the MII due to dependence constraints, see [8].

The MII due to resource constraints on a resource R is calculated by dividing the number of uses of the resource R by the number of resources of type R. The final MII is the maximum MII of all resource types. Before assignment, this number is calculated adding together similar resources from the MC and the RC. For example, the MII resource calculation can be seen on Table 2. In this case the MII is 2.

The assignment phase must assign, for a given iteration interval (II), the operations to either of the clusters, MC or

RC, while ensuring that there are enough resources to allow a schedule. Resources are the computation elements (processing elements and FFUs) as well as the RFU ports used to transfer data to and from the RFU. The RFU ports are modeled as a resource with zero latency during scheduling.

Table 2 $MII_{resource}$ calculation for the example

	Resources			Nodes	$MII_{resource}$
	Main cluster	Reconfigurable cluster	Total		
LD/ST	2	0	2	4	2
AU	1	6	7	9	2
LU	1	3	4	4	1
B	1	0	1	1	1
SHIFT	1	3	4	4	1

The loop will then be scheduled using a loop scheduler. Scheduling has been implemented using modulo scheduling [9]. If a schedule is found, the process is finished. If not, the II is incremented and a new assignment is performed.

4.1 RFU assignment

Once an iteration interval has been set, operations have to be assigned to a cluster. The assignment process is as follows:

1. Do initial assignment.
2. Add cluster-to-cluster transfers.
3. If assignment is valid, assignment successful.
4. If there are no nodes to mode, fail.
5. Move the best node from RC to MC and go to 2.

The initial assignment assigns to the RFU all operations that can be executed there. As the RFU contains more resources than the FFUs, this assignment is closer to the final solution than the opposite assignment. Operations that cannot be performed in the reconfigurable array are assigned to the main cluster. These operations are typically the branch operations, the floating-point operations and, when the reconfigurable unit does not have direct connection to the memory hierarchy, the memory access operations. As mentioned in section 3, operations that can be performed in the reconfigurable array can always be performed in the main cluster. The coloring in Figure 2 shows the previous example after initial assignment has been performed on the processor of Figure 1.

The assignment is then annotated with inter cluster transfers. These transfers are used to transfer data from the RFU to the FFUs or the other way round. The transfers require the usage of the RFU ports, which is considered a scheduling resource, just like the processing elements or the FFUs. Figure 3 shows the previous graph annotated with cluster-to-cluster transfers.

An assignment is only valid if there are enough resources to schedule the loop in the specified II (i.e. the number of uses of a type resource must be smaller than the number of resources of that type times the iteration interval). An assignment can be invalid, especially when the II is close to the MII, due to the following main reasons:

- Not enough communication resources between clusters: This happens when there are many data dependence chains going from one cluster to the other. The solution is to put complete chains in one of the clusters. As initially everything is assigned to the RC, the solution involves moving operations from the RC to the MC.
- Not enough resources in the reconfigurable cluster: All resources in the RFU are used while there are still some free on the main cluster. This can happen when the II is very close to the minimum iteration interval. The solution is to move operations that do not have enough resources to the main cluster since there should be free resources there.
- Not enough resources in the main cluster. Due to the nature of the assignment process, (always from RC to MC) this means that assignment is not feasible and the II must be incremented.

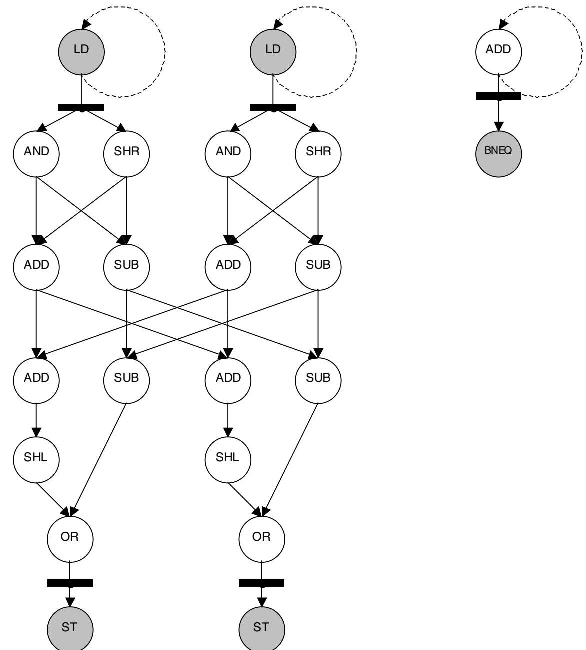

Figure 3 Data dependence graph with intercluster transfers shown as a thick line.

In order to see if the assignment is in one of the above cases, the MII is recalculated but this time including the data transfers between MC and RC and.

If this assignment is not valid for the given II, a node is moved from the RC to the MC. This process is repeated until a solution is found or there are no more nodes to be moved or if moving a node to the FFU causes a lack of resources in the MC.

A heuristic is used to choose which node to move. The heuristic calculates the modification the effect of the node movement on the MII of the assignment. The node that will produce the best change will be moved from the reconfigurable cluster to the main cluster. Any standard search algorithm can be used to find a solution. If no solution is found, the iteration interval is incremented by one and the assignment phase starts again with the initial assignment. If a solution is found, the graph is transferred to the software pipelining phase.

Figure 3 presents the initial assignment graph annotated with cluster-to-cluster transfers. Table 3 presents, for the previous example, the MII calculated for the inter cluster communication and Table 4 the MII calculated for the resources in each cluster. From the tables it can be seen that the communication from the RC to the MC is limiting the MII to 3, instead of 2 as was previously calculated. A search is done for a node to be moved from the RC to the MC. The ADD node connected to the branch operation is found and moved to the main cluster. This modification reduces the inter cluster communication by one. After this modification, the highest MII is 2. Modulo scheduling can now proceed.

Table 3 MII due to cluster-to-cluster communication

	Resources	Nodes	MII
MC to RC	2	2	1
RC to MC	1	3	3

Table 4 MII due to resource constraints

	Resources	Nodes	MII
MC, LD/ST	2	4	2
MC, AU	1	0	0
MC, LU	1	0	0
MC, B	1	1	1
MC, SHIFT	1	0	0
RC, AU	6	7	2
RC, LU	3	4	2
RC, SHIFT	3	4	2

4.2 Spatial computation

With the algorithm already described, it is possible to generate code for the reconfigurable processor but spatial computation is not exploited. Some modifications on the modulo scheduling algorithm need to be performed first. The main thing that has to be done is to exploit operations that have fractional latencies. On standard software pipelining, operations must begin and end in a cycle boundary (i.e. the latencies are always an integer number), which is not the case if spatial connections are performed.

In standard modulo scheduling, an operation can be scheduled at time T if:

1. It meets the dependence constraints, and
2. There are available resources for it to be scheduled

Meeting the dependence constraints for a given operation normally means that the operations that it depends on have been scheduled and finish in a previous cycle and that the operations that depend on this one are scheduled after the given operation produces its results. This approach is valid for operations with a latency that is an integer number.

In order to do spatial computation a modification to modulo scheduling has to be done. An operation with a fractional latency can start in a non-integer time. That is, the operation can begin at the end of an operation with fractional latency that started in the same cycle. In this case, the combined latency must be smaller than one cycle. This ensures that all spatial computations are synchronized with the temporal computations.

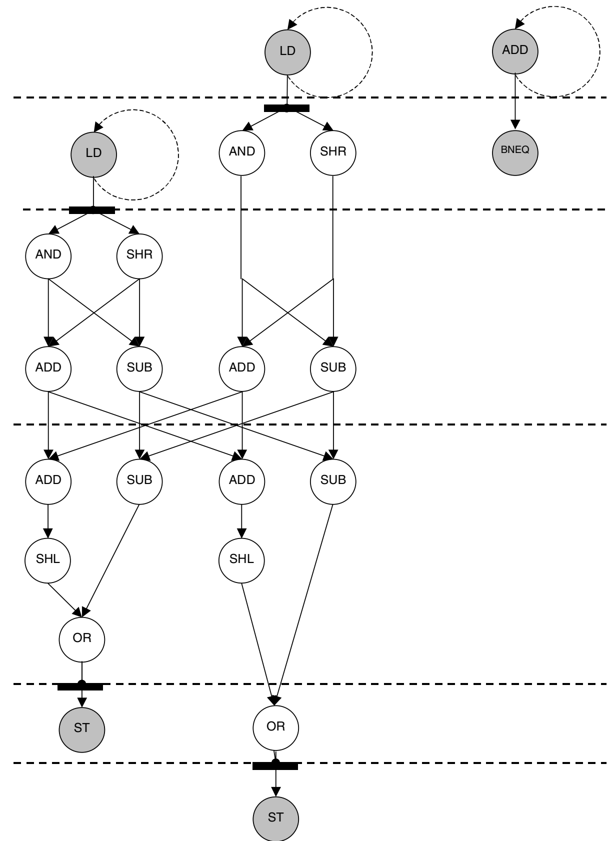

Figure 4 Graph divided in cycles

Taking this into account, an operation can be scheduled in the same cycle as operations that have dependences with it. The dependences must be checked at a sub-cycle

precision. Continuing with the example, Figure 4 presents the graph after performing modulo scheduling. As can be seen, spatial computation is exploited in cycles 3 and 4.

Table 5 presents the complete schedule. Cycles 1 to 4 are the prolog of the pipelined loop. Cycles 5 and 6 are the loop kernel. Cycles 7 to 10 are the epilog of the loop. The loop executes three iterations of the original loop at the same time. In Table 5, each color represents a different iteration.

After successfully scheduling the loop, the set of operations scheduled in the processing elements of the RFU for a cycle represent a new reconfigurable instruction. The number of instructions created will be equal to the iteration interval of the loop (i.e. two in the example, RFUOP1 at cycle 5 and RFUOP2 at cycle 6). The reconfigurable instruction can be seen as a compressed form of all the operations executed in the RFU.

5 Results

The algorithm presented here has been tested with the inner loop of an 8x8 DCT used in video and image compression. Each iteration performs an 8-point DCT. The target processor had 8 load/store units and 4 integer units in order to remove the effects of data transfers. The DDG of the inner loop has 102 nodes.

In order to test the assignment algorithm, the II interval of three processors were compared: a VLIW processor without RFU but with extra FUs (labeled VLIW), a processor with 2 RFU inputs and 1 RFU output (labeled 2,1) and a processor with 4 RFU inputs and 4 RFU outputs (labeled 4,4). The number of processing elements (or extra FUs) ranged from 1 to 32. Figure 5 shows the resulting data. Spatial computation was disabled in this experiment.

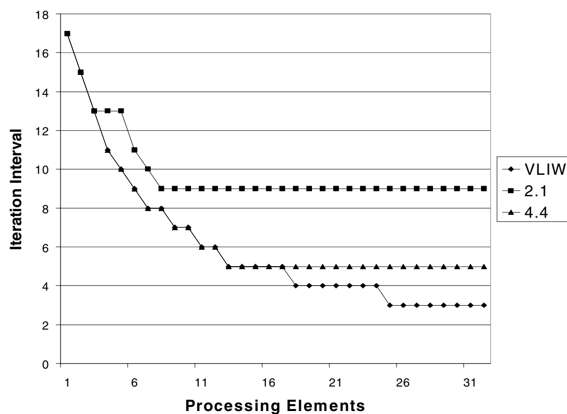

Figure 5 Assignment results

The VLIW line presents the ideal assignment case where there are enough RFU ports to make all communication. This is ideal since no VLIW processor would have that many functional units due to the size of the instruction word. As can be seen from the graph, both RFU curves almost match this ideal case until a diverging point where the II stops decreasing even when more processing elements are added. At this point, the communication resources impose a MII that cannot be decreased by adding more PEs. As expected, the processor with more RFU ports attains a better II (i.e. 5).

Figure 6 shows the total number of cycles spent on the loops of an H.263 decoder that can be software pipelined. As in the previous figure, adding more processing elements does not improve performance when there is a lack of communication resources.

Each loop has been independently compiled. During execution, every time a new loop is reenters, the RFU must be reconfigured. This is, effectively, dynamic reconfiguration. The time required for this reconfiguration has not been included in these results.

Figure 7 shows the effects of adding spatial computation to the two reconfigurable processors used in the previous measurements. From the figure we see that an improvement in execution speed of around 10%. This decrease in speed is due to a reduction in the prolog and epilog size. With higher number of iterations, the improvement would not be as good. The II remains the same since the MII is limited by resource constraints.

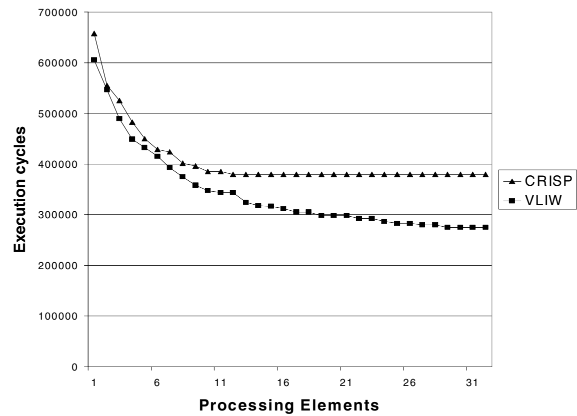

Figure 6 Cumulative assignment results on pipelinable loops of an H.263 decoder

6 Conclusions and Future Work

This paper has shown that software pipelining can be an effective technique for code generation for coarse-grained reconfigurable instruction set processors. The basic software pipelining has been extended with an assignment phase used to divide the code between the RFU and the normal function units. This has shown that a coarse-

grained reconfigurable instruction set processor is a viable alternative for very wide issue VLIW processors.

Software pipelining has also been enhanced by adding support for spatial computation. Spatial computation allows for a reduction in the number of execution cycles for loops with and without loop carried dependences, with the latter gaining the most benefit.

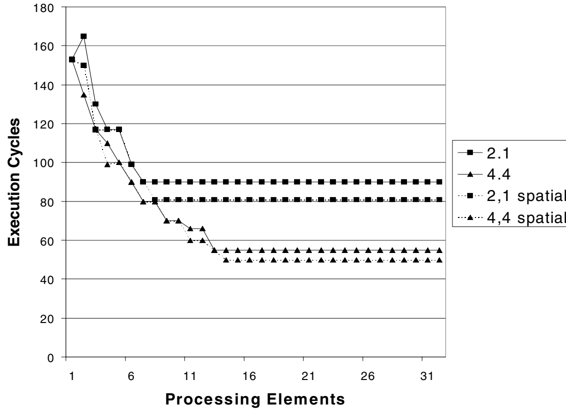

Figure 7 Spatial computation decreases cycle count

The presented technique combines instruction generation and instruction scheduling in a single algorithm, in contrast with other approaches that performs the two tasks in different steps. This fact allows the mapping of unconnected graphs onto the RFU, something not usually done.

Future work will focus on the problem of register allocation inside the RFU and changing the full crossbar to a more power efficient interconnect structure.

References

[1] M.F. Jacome and G. de Veciana, “Design Challenges for New Application-Specific Processors” IEEE Design & Test Computers, Vol. 17, No. 2, april 2000, pp. 40-50.

[2] F. Barat and R. Lauwereins, “Reconfigurable Instruction Set Processors: A survey”, Proceedings. 11th International Workshop on Rapid System Prototyping, 2000.

[3] André DeHon, “The Density Advantage of Configurable Computing”, IEEE Computer, Vol.33, No. 4, April 2000

[4] C. Alippi, W. Fornaciari, L. Pozzi and M. Sami, “A DAG-Based Design Approach for Reconfigurable VLIW Processors”, Proc. Of the IEEE Design and Test Conference in Europe, Munich, March 1999.

[5] T.J. Callahan and J. Wawrzynek, “Instruction Level Parallelism for Reconfigurable Computing”, Field-Programmable Logic and Applications, 8th International Workshop, September 1998.

[6] M. Budiu and S. Goldstein. “Fast compilation for pipelined reconfigurable fabrics”, Proceedings of the 1999 ACM/SIGDA FPGA '99, Monterey, CA, Feb. 1999, pp. 195-205.

[7] P. Op de Beeck, F. Barat, M. Jayapala, R. Lauwereins, "CRISP: A Template for Reconfigurable Instruction Set Architectures", Proceeding of Field Programmable Logic 2001.

[8] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan ,“Software pipelining,” ACM Computing Surveys, Vol. 27 No. 3, September 1995.

[9] Rau, B.R., “Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops”, MICRO-27. Proceedings of the 27th Annual International Symposium on Microarchitecture, 1994. Page(s): 63 – 74

Table 5 Final loop schedule and RFU instructions

	LD	LD	AU	LU	B	SH	AU	AU	AU	AU	AU	AU	LU	LU	LU	SH	SH	SH
1	LD		ADD															
2		LD			BNEQ													
3	LD		ADD				ADD	SUB	ADD	SUB			AND	AND		SHR	SHR	
4		LD			BNEQ		ADD	SUB	ADD	AUB			OR			SHL	SHL	
5	LD	ST	ADD				ADD	SUB	ADD	SUB			AND	AND	OR	SHR	SHR	
6	ST	LD			BNEQ		ADD	SUB	ADD	SUB			OR			SHL	SHL	
7		ST					ADD	SUB	ADD	SUB			AND	AND	OR	SHR	SHR	
8	ST						ADD	SUB	ADD	AUB			OR			SHL	SHL	
9		ST													OR			
10	ST																	

RFUOP1 =	ADD	SUB	ADD	SUB			AND	AND	OR	SHR	SHR	
RFUOP2 =	ADD	SUB	ADD	SUB			OR			SHL	SHL	