Loop Parallelization on a Reconfigurable Coprocessor

Erwan Fabiani, Dominique Lavenier, Laurent Perraudeau
IRISA
Campus universitaire de Beaulieu, Rennes, France
(efabiani,lavenier,perraude)@irisa.fr

Abstract— **The compilation of loops onto reconfigurable hardware is motivated mainly by two observations: the continuous growth of chip density, which will lead to new trends in micro-processor architecture, and the fact that time-critical parts of many applications are nested loops. This paper presents a compilation framework based on the idea that future microprocessors will have large built in reconfigurable area on which time-consuming loops will be parallelized as dedicated coprocessors made of regular arrays of identical hardwired processors.**

I. Introduction

In the next few years, chips will contain hundreds of millions of transistors: recently, IBM announced the development of a new manufacturing process, called CMOS 7S technology which uses copper instead of aluminum, and expects in the very near future to pack 150 to 200 million transistors on a single chip; Intel, AMD and Motorola consortium created to develop the extreme ultraviolet technology forecasts one billion transistors in the early years of the next century.

This transistor explosion raises the following question: "What are we going to do with chips containing a billion transistors ?" , or in other words: "How will microprocessor architectures make the best use of these riches ?". Burger and Goodman [3] draw the trends of processor architecture evolution from *advanced superscalar architectures*, which scale up from current designs to issue 16 or 32 instructions per cycle, to *RAW machines* (a MIT project) which seeks to "eliminate the instruction set as a binding contract between software and hardware". Such an architecture, without completely abandoning instruction sequencing, keeps a large place for reconfigurable hardware for tailoring specific bit or byte-level computation in a more powerful way than those supplied, for example, by MMX-like instruction sets. Undoubtedly, the next processor generations, thanks to the high transistor density, will include reconfigurable hardware areas dedicated to specific treatments from which parallelism and customization can be efficiently exploited. National Semiconductor's NAPA-1000 [2] or Motorola CORE+ [1] processors are both examples of such forerunner devices.

From a compiling point of view, we may wonder, first, about the portions of code which can benefit from these reconfigurable areas, and, second, how to derive custom hardware from high level programming languages.

It is well known that in many applications the critical time is spent in loops. Consequently, we should concentrate on these regions for speeding up computation, especially by extracting parallelism. Much research has already been done for parallelizing loops on general purpose parallel machines, and manually designed reconfigurable hardware implementations have demonstrated their efficiency. The success comes both from (1) the implementation of parallel loops on processor arrays which share, in common with the reconfigurable philosophy, regularity and locality, and (2) from the hardware optimization of the some inner loops having bit-level operations like masking, shifting and non-standard word sizes.

However designing manually optimized reconfigurable architectures remains a hard and tedious task. Today, it requires hardware competence, notably the use of VLSI CAD tools which is still the best way of programming reconfigurable components. From an HDL(Hardware Description Language) description, hardware compilers are able to generate bitstreams for such devices. But this is a VLSI-oriented process, which generally takes a very long time, and consequently becomes inappropriate if one intends to transparently include this approach into a compilation framework.

Because of the type of the architectures derived from nested loops (regular arrays), the mapping process can be drastically shortened by exploiting information relative to topology and regularity: for example, the place-and-route process can be strongly directed by the array topology, thus avoiding the simulating annealing placement step which is generally a very time-consuming task. In addition, criteria traditionally used in hardware compilers for optimizing reconfigurable resources – and which significantly slow down the mapping process – may not be, of the most importance here. Of course, high speed and minimal area must be a constant objective, but in a compiling context, it is certainly more important to get rapidly (in a few tens of seconds) a correct implementation providing a significant speed-up, than waiting hours for an optimized solution which can be changed at any time.

The rest of the paper is organized as follows. The next section briefly presents the overall approach for compiling loops onto a reconfigurable coprocessor. Section III and IV detail respectively two important steps: the partitioning and the physical mapping. The last section gives some conclusions.

II. Overall approach

The starting point of our approach is a sequential program with nested loops. The goal is to derive automatically a hardware description of the time-consuming loops, which will be supported by the reconfigurable hardware, and the complete hardware/software interface. No assumption is made on the way the reconfigurable area is linked to the processor: it can hang directly on the bus memory (as in the ArMen machine [5], [4]) or be attached through an I/O bus (as in the PAM engines [13]). We restrict the target architecture supporting the loop parallelization to a linear array of processors. The main reason is that more complex topologies, such as a 2D array, generally require very high data throughput which cannot be sustained by the processor.

With this architecture in mind, the compilation can be split into three main steps :

1 - Loop detection and parallelization on systolic arrays: This step detects time-consuming nested loops for which one can hope to speed-up the computation through specialization and parallelism. Detection can be performed on the basis of static analysis or profiling technics. Once the interesting loops have been selected, parallelization can occur. It consists of deriving regular array architectures (systolic as well as semi-systolic) from loop specifications or equivalent formal description such as systems of *affine recurrence equations* [10] : this model supports a powerful theory of space-time transformation methods and is now available for automatic parallelization as well as for derivation of systolic arrays. At IRISA, a functional language, ALPHA [7], based on systems of recurrence equations has been developed and a transformation system is used for exploring the transformations needed for systematic derivation of regular arrays.

2 - Partitioning: This step starts with a parallel description of the loops on a linear array. This can be an ALPHA program appropriately transformed or any other description (for instance, a description based on a data-parallel language). Since the available reconfigurable resources have physical limits and may not support the entire array, transformation of the architecture is required. It consists in splitting the array into subarray or clustering group of processors as explain in section III. The automatization of this task belongs still to the research domain and is not yet solved.

3 - Physical mapping: The last step maps the linear array on the reconfigurable area. A VLSI oriented approach is prohibited, due to the long time it requires to find optimized place and route solution. The key idea for speeding up the process is based on the two following points: (1) a definitive placement can be directly issued from the properties of regularity of the array; (2) a judicious placement implies a fairly short routing step. Preliminary experiments (see section IV) tend to demonstrate the soundness of these two assumptions: the mapping computation time is drastically shortened, rendering this method viable in a compiling approach.

The three steps are currently under investigation using two main approaches. The first one considers the ALPHA language which, through systematic transformations, covers step 1 and step 2. The second one consider a data-parallel language, C-stolic[11], which requires the parallelization of the loops to be performed by the programmer. Partitioning can also be explicitly specified in that language. The advantage of using C-stolic as an alternative approach is that steps which are not yet completely automatically solved can be performed manually, allowing immediately the test of the back end mapping tools.

III. Partitioning

This step transforms the parallel description of nested loops into a structural description. As a direct synthesis (one iteration producing one processor) could lead to a linear array with a large number of processors, a partitioning of the array is required to take care of the reconfigurable hardware capacity. Thus, the aim is to transform a description of a virtual array of P processors into a parameterized description of the same array using only K "real" processors, K being a parameter ($K \leq P$).

There are two ways of partitioning: LSGP (Locally Sequential, Globally Parallel) and LPGS (Locally Parallel, Globally Sequential) [8].

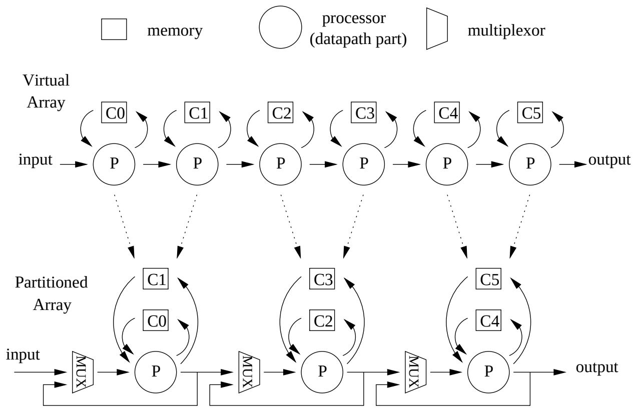

Fig. 1. The LSGP partitioning method : $P = 6, K = 3, Q = 2$

LSGP: the array of P virtual processors is divided in K groups of $Q = \lceil P/K \rceil$ processors. Each group is implemented in one "real" processor which executes sequentially Q iterations. The K processors work in parallel as a pipeline. As we can see on figure 1, the disadvantage of this method is the need to transform the description of the processor by adding a memory and a multiplexor.

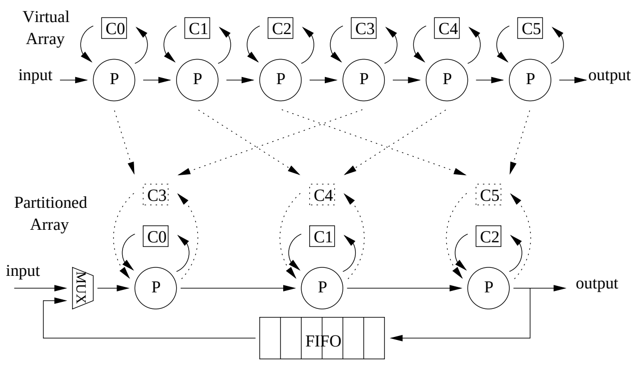

Fig. 2. The LPGS partitioning method : $P = 6, K = 3, Q = 2$

LPGS: the array of P virtual processors is divided in

$Q = \lceil P/K \rceil$ groups of K processors. Each group is implemented sequentially on the "real" array, as shown on figure 2. With the LPGS method, a processor does not need to be transformed: only an extra memory outside the array is required. On the other hand, an array with bidirectional communications can not be so partitioned.

Partitioning methods are expected to be automated with formal transformations, which can be implemented using the ALPHA system. Then, the result (a parameterized description of the array) can be synthesized by using the existing transformations. The synthesis produces an RTL (Register Transfer Level) description in the ALPHARD language [9] which can be directly used as an entry point for the physical mapping.

As partitioning is not yet automatic, we may use hand transformations on a C-stolic description. Currently, to give a RTL description to the physical mapping step, we are developing a compiler to transform a C-stolic program into a HDL (Hardware Description Language) description.

IV. Physical mapping

The last step is the physical mapping of regular arrays on FPGAs. The implementation of this step is based on respecting the regularity of regular arrays. We first explain why we have chosen to respect the regularity of regular arrays for the physical mapping. We then detail the steps of this physical mapping. Finally we present preliminary results which validates our assumptions.

A. Why implement regular arrays regularly?

In order to reduce the time spent in the usual place-and-route phase, we base the physical mapping step on a regular placement which relies on the following principles:
- signals which belong to a same processor of the regular array must be placed in a same region;
- identical processors must be implemented identically;
- neighboring processors of the regular array must be close on the FPGA circuit.

Our thesis is that a regular placement decreases the physical mapping time while allowing simpler place-and-route algorithms: this improves chances to find a routing for complex circuits. First of all, the placement phase will be faster since it is not calculated globally but simply deduced from the regular array architecture. In fact, as soon as a processor placement is found, it is enough to replicate it for other processors. The time saving for the routing level is linked to the regular placement. Since the CLBs (Configurable Logic Blocks) which implement a processor are confined to a same zone, the number of possible paths between CLBs is reduced and the router will be able to more easily and quickly connect the CLBs. Moreover, the router is likely to find the shortest path between CLBs as well as between processors.

B. Implementation steps

The physical mapping is globally divided into four steps: area estimation, placement of the array, placement of the processors and routing.

1 - Area estimation: This step predicts: (1) what resources (counted in CLBs, logical gates, LUT (Look Up Table) or registers) are required for the implementation of one processor on the targeted FPGA; (2) what are the minimal dimensions of one processor area.

2 - Placement of the array: This step consists of allocating geometric blocks of CLBs for each processor and specifying relative placement between these blocks. In this step, we use the principle that neighboring processors in the regular array must also be neighbors on the FPGA circuit. Moreover, the zones where to place the first and the last processor are determined according to the location of I/0 ports. To satisfy these constraints, segments of the linear array are folded up, to obtain a snake like form (Fig. 3), which can be mapped to a grid of CLBs. Several crite-

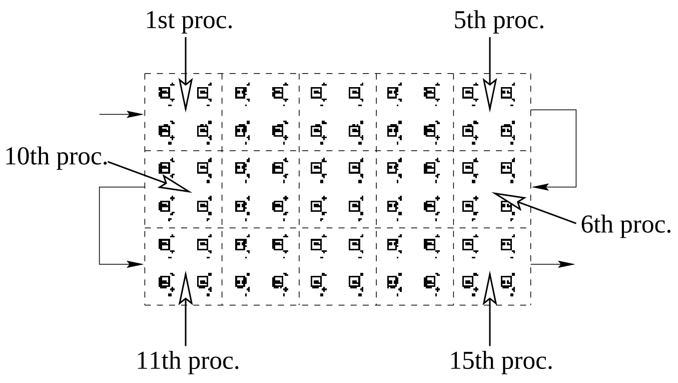

Fig. 3. Mapping of a regular array of 15 processors (4 CLBs per processor) following a snake like form

ria can guide the array placement and select the solutions: maximum number of processors implemented, maximum area allocated to the processors, maximum space between neighbors processors or minimum area used for the array. When this step is complete, dimensions of the area allocated to processors and relative placement of these processors are known.

3 - Processor placement: Each processor is placed respecting the processor area dimensions found in the previous step. As we assume that the structure of one processor is described by an assembly of basic operators, several placement methods could be used. The two extreme methods are:
- copying the processor structure onto the FPGA circuit, i.e. having a placed description for each operator of the processor and placing them in the zone allowed to one processor, respecting the data path between those operators. This method tends to minimize the use of routing resources, but the lose of area may be too high in the case of coarse grain FPGA circuits;
- translating the processor description in terms of boolean equations, to logically optimize those and to map and place them into the area allowed to the processor (method used for our experimentations). In this case we obtain the minimal area for one processor. But this method requires high routing resources and may take too much time if the individual processors are too large.

The first method would better fit for fine grain FPGA circuits (like XC6200 [16]) whereas the second one is more adapted to coarse grain FPGA circuits (like XC4000 [15]). The best method, situated between these two extremes, should be determined according to the granularity of the targeted FPGA, the complexity of the processors and the degree of utilization of standard tools.

4 - Routing: This last step achieves the routing of the whole design. Here standard tools are making the routing. The regular placement found in the previous step induces a short time routing phase.

C. Experimentations

Some experimentations have been conducted for confirming our assumptions on the advantages of regular placement. The targeted FPGA family for those experimentations is the Xilinx XC4000 family. The placer-router used is Xilinx PPR[14]. To specify the regular placement we use the PamDC language[12]. Two regular circuits were tested:

- a Lyon's bit-serial multiplier [6] which uses integers whose word-length (in bits) is equal to the number of processors of the array;
- a convolution array whose multiplier is replaced by a parallel AND operation; this circuit has been tested for two word-lengths: 8 and 16 bits.

On the FPGA circuit the arrays are placed following a snake like form. For the internal placement (and partitioning) of the processor we used a simple method: this is done using PPR with constraints permitting to obtain the minimal area (in CLBs) for one processor. Then the partitioning and placement found by PPR are recovered, and following those, the internal description of the processors is rewritten in PamDC. For our tested circuits, this preliminary use of PPR takes no more than 1 minute. Moreover, the area occupied by circuits placed with our method is equal to that of circuits entirely placed by PPR. The measured mapping time is the execution time (on a 140 MHz Sun Ultra1 workstation) of PPR for each circuit. In one case the circuit is entirely partitioned, placed and routed by PPR. In the other case, the circuit is pre-placed regularly and the sole job of PPR consists in routing it. As one can see (table I), the time savings induced by a regular placement is significant. Moreover, theoretical clock frequencies of regularly placed circuits were faster by 8 to 25%. Although these preliminary tests are not generalizable and the internal placement processor method is specific to coarse grain FPGA, this conforms to the idea that regular arrays must be placed regularly on FPGA circuits.

TABLE I
MAPPING TIME (IN MIN, SEC) OF CIRCUITS IMPLEMENTED ON A XC4020(784 CLBS), REGULARLY PLACED OR NOT: LYON'S MULTIPLIER, 8 BITS CONVOLUTION AND 16 BITS CONVOLUTION.

circuit	regular placement		reduction
	no	yes	(%)
lyon	24,21	4,25	82
conv8	36,37	5,01	86
conv16	42,43	11,17	74

V. Conclusion

The compilation framework we are currently developing aims to provide tools for the next generation of microprocessors. We expect that such devices will include a reconfigurable zone on which dedicated coprocessors could be implemented for speeding up time-consuming portion of code. We believe that the success of such architectures will greatly depend on the ease and the power of programming tools, and that, right now, advanced compiling tools have to be thought out. However, even if this work is guided towards tomorrow's architectures, it can be largely applied to existing architectures which already incorporate reconfigurable components and for which no high level programming environment is provided.

References

[1] http://www.mot.com/sps/hpesd/prod/coldfire/core_fact.html.

[2] http://www.national.com/appinfo/milaero/napa1000.

[3] D. Burger and J.R. Goodman. Billion-transistor architecture. *Computer*, 30(9):46–50, September 1997.

[4] J. Champeau, L. Le Pape, B. Pottier, S. Rubini, E. Gautrin, and L. Perraudeau. Flexible parallel fpga-based architecture with armen. In *HICSS'94*, Hawai, January 1994.

[5] D. Lavenier, B. Pottier, F. Raimbault, and S. Rubini. Fine grain parallelism on a MIMD machine using FPGAs. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 2–8, Napa, CA, USA, April 1993. IEEE Computer Society Press.

[6] R.F. Lyon. Two's complement pipeline multipliers. *IEEE Transactions on Communications*, pages 418–425, April 1976.

[7] C. Mauras. *Alpha : un langage équationnel pour la conception et la programmation d'architectures systoliques.* PhD thesis, Université de Rennes 1, December 1989. A partial english version can be find at http://www.ee.byu.edu/~wilde/Alpha/.

[8] D.I. Moldovan and J.A.B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transactions on Computers*, 35(1):1–12, January 1986.

[9] P. Le Moënner, L. Perraudeau, S. Rajopadhye, T. Risset, and P. Quinton. Generating regular arithmetic circuits with alphard. In *Massively Parallel Computing Systems (MPCS'96)*, Ischia, Italie, May 1996.

[10] P. Quinton and V. V. Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1989.

[11] F. Raimbault and D. Lavenier. Relacs for systolic program. In *ASAP'93: International Conference on Application Specific Array Processors*, Venice, Italy, October 1993.

[12] H. Touati. Pamdc: a c++ library for the simulation and generation of xilinx fpga designs. Technical report, Digital Equipment Corporation Systems Research Center, 1997.

[13] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable Active Memories: Reconfigurable Systems Come of Age. *IEEE Transactions on VLSI Systemes*, 4(1), March 1996.

[14] Xilinx. *XACT Reference Guide, Vol II: PPR*, 1994.

[15] Xilinx. *The Programmable Logic Data Book*, 1997.

[16] Xilinx. *XC6200 Field Programmable Gate Arrays*, April 1997.