REGULAR MAPPING FOR COARSE-GRAINED RECONFIGURABLE ARCHITECTURES

Frank Hannig, Hritam Dutta, and Jürgen Teich

Department of Computer Science 12, Hardware-Software-Co-Design,
University of Erlangen-Nuremberg, Germany

ABSTRACT

Similar to programmable devices such as processors or micro controllers also reconfigurable logic devices can be built as software, by programming the configuration of the device. In this paper, we present an overview of constraints which have to be considered when mapping applications to coarse-grained reconfigurable architectures.

The application areas of most of these architectures addressing computational-intensive algorithms like video and audio processing or wireless communication. Therefore, reconfigurable arrays are in direct competition with DSP processors which are traditionally used for digital signal processing. Hence, existing mapping methodologies are closely related to approaches from the DSP world. They try to employ pipelining and temporal partitioning but they do not exploit the full parallelism of a given algorithm and the computational potential of typically 2-dimensional arrays. We present a first case study for mapping regular algorithms onto reconfigurable arrays by using our design methodology which is characterized by loop parallelization in the polytope model. The case study shows that our regular mapping methodology may lead to highly efficient implementations taking the constraints of the architecture into account.

1. INTRODUCTION

These days, semiconductor technology allows implementing arrays of hundreds of 32-bit microprocessors on a single die and more. Computationally intensive applications like video and image processing in consumer electronics and the rapidly evolving market of mobile and personal digital devices are the driving forces in this technology. The increasing amount of functionality and adaptability in these devices has lead to a growing consideration of coarse-grained reconfigurable architectures which provide the flexibility of software combined with the performance of hardware. But, on the other hand, there is the dilemma of not being able to focus the hardware complexity of such devices because of a lack of mapping tools. Hence, parallelization techniques and compilers will be of utmost importance in order to map computationally intensive algorithms efficiently to these coarse-grained reconfigurable arrays. In this context, our paper deals with the specific problem of mapping a certain class of regular nested loop programs onto a dedicated processor array. This work can be classified to the area of loop parallelization in the polytope model [1].

The authors would like to thank PACT XPP Technologies for their collaboration and support.

The rest of the paper is structured as follows. In Section 2, a brief survey of previous work on reconfigurable computing is presented. Section 3 introduces our design flow. In Section 4, we give an overview which architecture constraints have to be taken into account during compilation of algorithms onto coarse-grained reconfigurable architectures. Afterwards in Section 5, a case study of our mapping methodology is given and results are discussed. Future work and concluding remarks are presented in Section 6.

2. RELATED WORK

Reconfigurable architectures span a wide range of abstraction levels from fine-grained LUT based reconfigurable logic devices, like FPGAs, to distributed and hierarchical systems with heterogeneous reconfigurable components. A taxonomy of reconfigurable logic devices is given in [2]. In order to handle the routing area overhead of FPGAs and herewith configurations' size another approach are coarse-grained reconfigurable architectures. Several academic coarse-grained reconfigurable arrays have been developed [3] and, since a while, more and more commercial are being developed like the D-Fabrix [4], the DRP from NEC [5], or the PACT XPP [6].

Both fine- and coarse-grained architectures have a lack of programmability in common, due to their own paradigms which are totally different from the *von Neumann's*. To overcome this obstacle some research work exists. For instance, the Nimble framework [7] for compiling applications specified in C to an FPGA. A regular mapping methodology for mapping nested loop programs onto FPGAs is also presented in the work of [8] and will be later described in this paper.

Only few research work is published which deals with the compilation to coarse-grained reconfigurable architectures. The authors in [9] describe a compiler framework to analyze SA-C programs, perform optimizations, and automatically map the application onto the MorphoSys architecture [10], a row-parallel or column-parallel SIMD architecture. This approach is limited since the order of the synthesis is predefined by the loop order and no data dependencies between iterations are allowed.

Another approach for mapping loops onto coarse-grained reconfigurable architectures is presented by Dutt et al. in [11]. Outstanding in their compilation flow is the target architecture, the DRAA, a generic reconfigurable architecture template which can represent a wide range of coarse-grained reconfigurable arrays. The mapping technique itself is based on loop pipelining and partitioning of the program

tree into clusters which can be placed on a line of the array. In this paper we present a case study based on mapping of regular algorithms and loop parallelization in the polytope model. The advantage of this approach is the exploitation of full data parallelism and that an efficient algorithm mapping in terms of space and time may be directly derived from the polytope model.

3. DESIGN FLOW FOR REGULAR MAPPING

In this section we give an overview of our existing mapping methodology PARO [8] when generating synthesizable descriptions of massively parallel processor arrays from regular algorithms. The main transformations during the design flow are briefly described in the following.

Starting from a given nested loop program in a sequential high-level language (subset of C) the program is parallelized by data dependence analysis into *single assignment code* (SAC). On an intermediate representation of equations and index spaces several combinations of parallelizing transformations in the polytope model can be applied:

Affine Transformations, like skewing of data spaces or the embedding of variables into a common index space.

Localization of affine data dependencies to uniform data dependencies by propagation of variables from one index point to a neighbor index point.

Operator Splitting, equations with more than one operation can be split into equations with only two operands.

Exploration of Space-Time Mappings. Linear transformations are used as *space-time mappings* in order to assign a processor (space) and sequencing index (time) to index vectors. Since, the number of possible space-time mappings can be very huge an efficient design-space exploration is performed. During this multi-objective optimization problem, latency as a measure of performance, area cost, and energy consumption can be considered [12, 13].

Partitioning. In order to match resource constraints such as limited number of processing elements, partitioning techniques have to be applied [14].

Control Generation. If the functionality of one processing element can change over the time control mechanism are necessary. Further control structures are necessary to control the internal schedule of a PE.

HDL Generation & Synthesis. Finally after all the refining transformations a synthesizable description in a hardware description language like VHDL may be generated. This is done by generating one PE and the repetitive generation of the entire array.

4. CONSTRAINTS OF COARSE-GRAINED RECONFIGURABLE ARCHITECTURES

Full custom ASIC designs have the advantage that they are only constrained by technological and economic parameters. Since programmable devices and reconfigurable logic devices are predetermined and limited in terms of resources, even if they are given as IP (intellectual property) model. In the following we outline which constraints have to be taken into account when mapping algorithms onto homogeneous coarse-grained (re)configurable architectures. Briefly, such an architecture consists of an *array* of *processor elements*, *memory*, *interconnect structures*, *I/O-ports*, *synchronization* and *reconfiguration mechanisms*.

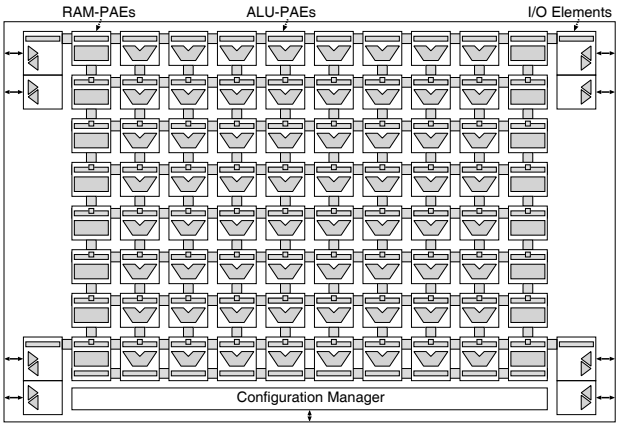

Fig. 1. Structure of the PACT XPP64-A processor.

Array. The size of the processor array. How many processor elements are in the array and how are they aligned, as one line of processing elements, several lines of processing elements, or one array of $N \times M$ processing elements.

Processor element. Each processor element contains resources for functional operations. This can be either one or more dedicated functional units like (multipliers, adders, etc.) or one or more arithmetic logic units (ALUs). In case of an ALU we have to consider if this unit is configured in advance or during a reconfiguration phase, or if this ALU can be programmed with an instruction set. In case of programmability it has to be considered if a local program is only modulo sequentially executed by a sequencer or if the instruction set includes also conditional branches.

Memory. Memory can be divided into local memory in the form of register files inside each processor element and into memory banks with storage capacities in the range from hundreds to thousands of words. The alignment and the number of such memory banks are important for the data mapping. In addition, knowledge of several memory modes is helpful, e.g., configuration as FIFO. If a processor element contains an instruction programmable ALU, besides the internal register file also an instruction memory is given.

Interconnect. Here, the structure and number of communication channels is of interest. Which type of interconnect is used, buses or point-to-point connections? How are these channels aligned, vertically, horizontally, or in both directions? How long can point-to-point connections be, without delay, or how many cycles have to be taken into account when communicating data from processor element P_{x_1,y_1} to processor element P_{x_2,y_2}? Additionally, similar structures are required to handle the control flow.

Synchronization. Whether the synchronization is explicit or implicit like in a packet-oriented dataflow architecture.

I/O-ports. The maximum bandwidth is defined by the number and width of the I/O-ports. The placement of I/O-ports is important, since they are responsible for feeding data in and out. Furthermore, it has to be considered if the I/O-port is a streaming port or an interface to external memory.

Reconfiguration. Here, the configuration time and the number of configuration contexts have to be considered. In ad-

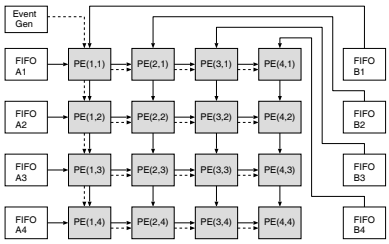

Fig. 2. 4×4 processor array.

dition, possibilities of partial and dynamical reconfiguration during the execution have to be considered.

5. CASE STUDY

In this case study our regular mapping methodology has been applied for a matrix multiplication algorithm. Here, different solutions are discussed and compared to existing results. But first, in the next subsection a PACT XPP64-A processor is described that is used as the target architecture of our case study. Afterwards, our mapping methodology is applied.

The PACT XPP64-A Reconfigurable Processor. PACT's XPP64-A [6, 15] is a high performance runtime reconfigurable processor array, a schematic diagram of the internal structure is shown in Fig. 1. The XPP64-A contains 64 ALU-PAEs (processing array elements) of 24 bit data with in an 8×8 array. RAM-PAEs are located in two columns at the left and right border of the array. The dual ported RAM has two separate ports for independent read and write operations. Furthermore, the RAM can be configured to FIFO Mode (no address inputs needed). Each of the 16 RAM-PAEs has a 512×24 bit storage capacity. Four independent I/O interfaces in the corners of the array provide high bandwidth connections from the internal data path to external streaming data or direct access to external RAMs.

Matrix Multiplication Algorithm. In the following regular mapping study we consider a matrix multiplication algorithm as given by the C-program (Fig. 3). It is assumed that

```
for (i = 0; i<N; i++)
{ for (j = 0; j<N; j++)
  { for (k = 0; k<N; k++)
    { c[i][j] = a[i][k] * b[k][j] + c[i][j];
    }
  }
}
```

Fig. 3. Matrix multiplication algorithm, C-Code.

the input matrices A and B are stored already in the arrays `a[N][N]` and `b[N][N]`, and that the arrays' elements of `c` are initialized with zero. After performing successively parallelization, embedding and localization of variables the algorithm in Fig. 4 is derived. The matrices A and B are embedded into the arrays as follows, `a[i][0][k]`$= a_{ik}$, `b[0][j][k]`$= b_{kj}$. Considering the amount of allocated memory it is obvious that resources are wasted by the C-program. On the other hand, full data parallelism is explicitly represented by this program. Furthermore, the algorithm is regular since affine data dependencies have been transformed to uniform data dependencies by localization.

In order to map the matrix multiplication algorithm to the 2-dimensional PACT XPP array a design space exploration [12] of possible space-time mappings is performed. This exploration gives a set of optimal mappings in terms of number of PEs and in terms of latency. From this set of optimal mappings we have selected one mapping. In the following case we consider a matrix of size $N = 4$. The processor array for the chosen mapping is shown in Fig. 2. Due to the localization procedure the matrix elements can be read into the array at the edges. Therefore, the matrix A is sequentially fed from the left side into the array where one row of the matrix is stored in one separate FIFO. The matrix B is sequentially fed row by row from the top into the array. Each column of B is stored also in one separate FIFO. Afterwards, the elements of A (B) are propagated cycle by cycle to the right (lower) neighbor processor element. Similar to this data propagation, also an event signal is propagated to reset every fourth clock cycle the accumulators of the processor elements. This is the same rate as the pipelinerate of the whole array, because after a delay of seven clock cycles every fourth cycle a new matrix multiplication can be performed. So the utilization of the 16 PAEs is 100% since in every cycle 16 Multiply and Accumulate (MAC) operations are completed. Since the result matrix C is distributed over all 16 processor elements, the readout of the result data has to be serialized. Due to space limitations we omit this transformation in this paper.

```
for (i = 1; i<N+1; i++)
{ for (j = 1; j<N+1; j++)
  { for (k = 1; k<N+1; k++)
    { a[i][j][k] = a[i][j-1][k];
      b[i][j][k] = b[i-1][j][k];
      z[i][j][k] = a[i][j][k] * b[i][j][k];
      c[i][j][k] = c[i][j][k-1] + z[i][j][k];
    }
  }
}
```

Fig. 4. Matrix multiplication algorithm after parallelization, operator splitting, embedding, and localization.

Partitioned Implementation. The example of the 4×4 matrix multiplication algorithm benefits from the parallel access to several internal memory blocks which have to be very closely coupled to a surrounding architecture, e.g., to a cache. Considering a more realistic scenario with three external memory blocks for each of the matrices A, B, and the result matrix C. Each memory block can be accessed via an independent I/O-port, i.e., in one clock cycle one element of matrix A, one of B can be read, and one of C can be written. From this follows that the reading of one column of A or one row B, respectively, takes 4 clock cycles and a MAC operation is performed only every fourth cycle per PE. Due to the I/O bottleneck, the processor elements (the PEs which are responsible for the arithmetic part of the calculation) are only utilized by 25%. To increase the utilization of the PEs the number is reduced by 75% to four PEs. In order to map the 4×4 matrix multiplication algorithm to the reduced number of PEs the algorithm has to be partitioned. When large problems have to mapped which do not fit onto the array this technique has to be applied. In Fig. 5 the partitioning scheme chosen here and the resulting processor array is depicted. In this example, an LPGS (local parallel, global sequential) partitioning scheme has been used. In this case, each index point within one tile corresponds to one physical processor that executes the index points at the same position of other tiles sequentially. The sequential execution order is depicted by the numbers inside the tiles. In

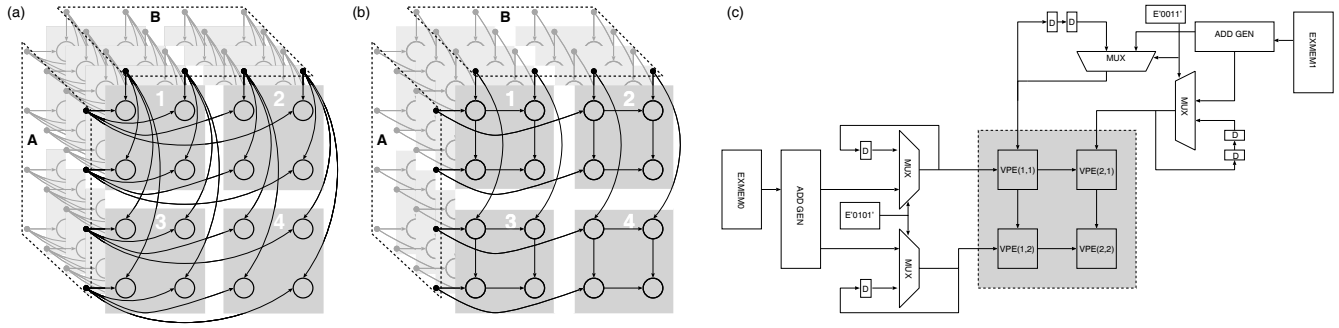

Fig. 5. (a), dataflow graph of the LPGS-partitioned matrix multiplication 4×4 example. (b), dataflow graph after performing localization inside each tile. (c), array implementation of the partitioned example.

order to save resources (local registers) firstly, the partitioning is performed, Fig. 5 (a), and afterwards the localization, Fig. 5 (b) [14]. The 2×2 processor array is again regular and similar to the architecture of the non-partitioned example. The sole difference is that for the accumulation of the variable c, a shift register of length four is necessary in each PE. The input to the virtual processor elements are taken internally from other PEs or externally from multiplexers as shown in Fig. 5 (c). The multiplexers are used to select delayed or incoming data from the external memory according to the event generators. The address generators are used to access the matrix coefficients from external memory. The implementation of the partitioned version requires a different addressing scheme which is stored as LUT in an internal memory.

Results and Comparison. In both of our implementations, the fullsize and the partitioned version of the matrix-matrix multiplication with input matrices of dimension 4×4, optimal utilization of resources is observed, i.e., each configured MAC-unit performs in each cycle one operation. Comparing our 4×4 array implementation with the matrix-vector multiplication as implemented by Gunnarsson et al. [16], it is observed that using fewer resources and with better implementation more performance per cycle can be achieved. The number of ALUs is reduced from $O(3N)$ to $O(N)$. The comparison was done by implementing our matrix-matrix multiplication as N matrix-vector multiplication. Furthermore, in our implementation the output serialization, i.e., merging and writing of output data streams is overlapped with computations in PEs, so we have no overhead associated with writing the data. This is a considerable improvement from the implementation presented in [16] where overhead associated with merging and writing data is $O(\frac{N}{2})$. Therefore, the computational time (block pipelining period) is reduced from $O(3N)$ to $O(N)$.

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented a mapping methodology based on loop parallelization in the polytope model in order to map nested loop kernels onto coarse-grained reconfigurable arrays for the first time. The obtained results are efficient in terms of utilization of resources and execution time. In the future we would like to adapt our PARO design methodology to coarse-grained reconfigurable arrays in order to perform automatic compilation of nested loop programs.

7. REFERENCES

[1] P. Feautrier, "Automatic Parallelization in the Polytope Model," Tech. Rep. 8, Laboratoire PRiSM, Université des Versailles St-Quentin en Yvelines, Versailles Cedex, June 1996.

[2] P. Schaumont, I. Verbauwhede, K. Keutzer, and M. Sarrafzadeh, "A Quick Safari Through the Reconfiguration Jungle," in *38th Design Automation Conference*, Las Vegas, NV, June 2001, pp. 172–177.

[3] R. Hartenstein, "A Decade of Reconfigurable Computing: A Visionary Retrospective," in *Proceedings of Design, Automation and Test in Europe*, Munich, Germany, Mar. 2001, pp. 642–649.

[4] Elixent Ltd., `www.elixent.com`

[5] M. Motomura, "A Dynamically Reconfigurable Processor Architecture," in *Microprocessor Forum*, CA, 2002.

[6] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "PACT XPP – A Self-Reconfigurable Data Processing Architecture," *The Journal of Supercomputing*, vol. 26, no. 2, pp. 167–184, 2003.

[7] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood, "Hardware-software Co-Design of Embedded Reconfigurable Architectures," in *37th Design Automation Conference*, Los Angeles, CA, June 2000, pp. 507–512.

[8] PARO Design System Project, `www12.informatik.uni-erlangen.de/research/paro`

[9] G. Venkataramani, W. Najjar, F. Kurdahi, N. Bagherzadeh, W. Böhm, and J. Hammes, "Automatic Compilation to a Coarse-grained Reconfigurable System-on-Chip," *ACM Trans. on Embedded Computing Systems*, to appear November 2003.

[10] H. Singh, M.-H. Lee, G. Lu, N. Bagherzadeh, F.J. Kurdahi, and E.M. Chaves Filho, "Morphosys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, 2000.

[11] J. Lee, K. Choi, and N. Dutt, "An Algorithm for Mapping Loops onto Coarse-grained Reconfigurable Architectures," in *Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, San Diego, CA, June 2003, pp. 183–188.

[12] F. Hannig and J. Teich, "Design Space Exploration for Massively Parallel Processor Arrays," in *Parallel Computing Technologies, 6th International Conference, PaCT 2001, Proceedings*, V. Malyshkin, Ed., Novosibirsk, Russia, Sept. 2001, vol. 2127 of *Lecture Notes in Computer Science (LNCS)*, pp. 51–65.

[13] F. Hannig and J. Teich, "Energy Estimation of Nested Loop Programs," in *Proceedings 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2002)*, Winnipeg, Manitoba, Canada, Aug. 2002.

[14] J. Teich and L. Thiele, "Exact Partitioning of Affine Dependence Algorithms," in *Embedded Processor Design Challenges*, E.F. Deprettere, J. Teich, and S. Vassiliadis, Eds., Mar. 2002, vol. 2268 of *Lecture Notes in Computer Science (LNCS)*, pp. 135–153.

[15] PACT XPP Technologies, *XPP64-A1 Reconfigurable Processor – Datasheet*, Munich, Germany, 2003.

[16] F. Gunnarsson, C. Hansson, D. Johnsson, and B. Svensson, "Implementing High Speed Matrix Processing on a Reconfigurable Parallel Dataflow Processor," in *Second International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'02)*, Las Vegas, NV, June 2002, pp. 74–80.