In *Proceedings of the Fourth International Workshop on Applied Reconfigurable Computing (ARC)*, London, United Kingdom, March 26-28, 2008.
Volume 4943 of Lecture Notes in Computer Science (LNCS), pp. 287-293, © Springer, 2008.

PARO: Synthesis of Hardware Accelerators for Multi-Dimensional Dataflow-Intensive Applications

Frank Hannig, Holger Ruckdeschel, Hritam Dutta, and Jürgen Teich

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

Abstract. In this paper, we present the PARO design tool for the automated hardware synthesis of massively parallel embedded architectures for given dataflow dominant applications. Key features of PARO are: (1) The design entry in form of a compact and intuitive functional programming language which allows highly parallel implementations. (2) Advanced partitioning techniques are applied in order to balance the trade-offs in cost and performance along with requisite throughputs. This is obtained by distributing computations onto an array of tightly coupled processor elements. (3) We demonstrate the performance of the FPGA synthesized hardware with several selected algorithms from different benchmarks.

1 Introduction and Related Work

The rising complexity of embedded digital applications and the growing importance of time-to-market require powerful modeling methods and tools to automate the design and implementation process. Whereas software compilers have reached a mature level, there still exist only few and restricted tools for the synthesis of hardware implementations from high-level algorithm descriptions. Commercial examples of such systems are Catapult-C from Mentor Graphics [9], Forte Cynthesizer [4], or PICO Express by Synfora [11]. Apart from commercial systems, there exist several C-based synthesis approaches for reconfigurable systems in academia. For instance, the SPARK [6] synthesis methodology which is particularly targeted to control-intensive signal processing applications. However, SPARK can handle only one dimensional arrays. The aforementioned design tools start from a subset of C, C++, or SystemC code. However, starting with sequential languages has the disadvantage that their semantics force a lot of restrictions on the execution order of the program. Most of the parallelism contained in the original mathematical model of the algorithm is lost during the transformation to sequential code. One option is to directly start from a functional language as for instance Haskell [13]. However, also Haskell has only restricted abilities to handle true multi-dimensional arrays (i.e., arrays in which every dimension is treated as equivalent). Other approaches try to avoid the restrictions of sequential languages by using different programming and execution models. For instance, the MMAlpha system [5] is based on loop parallelization in the polytope model similar to our approach. In previous work, we used our methodology only for handcrafted mapping of certain algorithms [3] or presented the generation of dedicated FPGA hardware accelerators for one parameterizable digital signal processing application [10]. Whereas, in this paper we present for the first time the PARO design tool for the automated generation of highly parallel hardware accelerators for a broad variety of multi-dimensional dataflow dominant applications selected from different benchmarks.

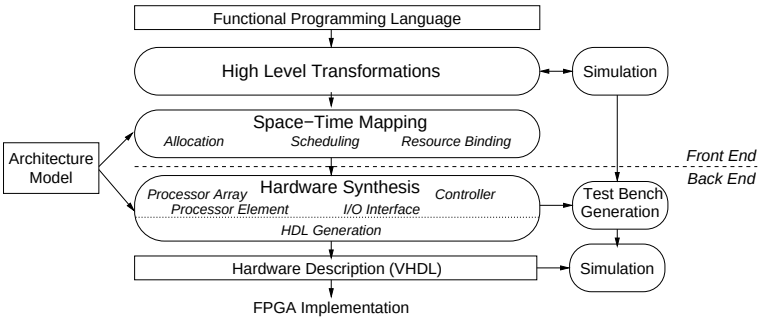

Fig. 1. PARO design flow trajectory for the generation of hardware accelerators.

2 Description of the Front End

An overview of our tool's design flow is depicted in Fig. 1. As design entry we developed a functional programming language. The class of algorithms that can be expressed by a PARO program is based on the mathematical model of *dynamic piecewise linear/regular algorithms (DPLA)* [7]. The language consists of a set of recurrence equations defined for a multi-dimensional iteration space as it occurs in nested loop programs. When modeling signal processing algorithms, a designer naturally considers mathematical equations. Hence, the programming is very intuitive. To allow irregularities in a program, an equation may have iteration and run-time dependent conditionals. Furthermore, big operators (also often called reductions) which implement mathematical operators such as $\sum$ or $\prod$ can be used. In contrast to the common mathematical notation, the iteration space is not required to be 1-dimensional, as the following image processing code fragment of a 2-D Gaussian window filter demonstrates

```
w[0,0] = 1; w[0,1] = 2; w[0,2] = 1;
w[1,0] = 2; w[1,1] = 4; w[1,2] = 2;
w[2,0] = 1; w[2,1] = 2; w[2,2] = 1;
h[x,y] = SUM[i>=0 and i<=2 and j>=0 and j<=2](pic_in[x+i,y+j] * w[i,j]);
pic_out[x,y] = h[x,y] >> 4;  // divided by 16
```

Based on a given algorithm, various source-to-source compiler transformations and optimizations can be applied within the design system. Among others, these transformations include: Constant and variable propagations, common sub-expression elimination, loop perfectization, dead-code elimination, affine transformations of the iteration space, strength reduction of operators (usage of shift and add instead of multiply or divide), and loop unrolling. Algorithms with non-uniform data dependencies are usually not suitable for mapping onto regular processor arrays as they result in expensive global communication or memory. For that reason, a well known transformation called *localization* exists which replaces affine dependencies by regular dependencies.

For such a regular algorithm, the data dependencies can be visualized as reduced dependence graph (RDG), which contains one node for each variable. If there are several equations defining the same variable due to iteration dependent conditionals, there is only one node for the corresponding variable in the graph. So, the terms variable and node may be used synonymously in the following. The data dependencies are annotated to the edges. Figure 2 shows the RDG after localization of data dependencies of an FIR filter initially described by the equation $Y(i) = \sum_{j=0}^{N-1} A(j) \cdot U(i-j)$ with $0 \leq i < T$, N denoting the number of filter taps, $A(j)$ the coefficients, $U(i)$ the filter input, and $Y(i)$ the filter result.

Partitioning is a well known transformation which covers the index space of computation using congruent hyperplanes, hyperquaders, or parallelepipeds called *tiles*.

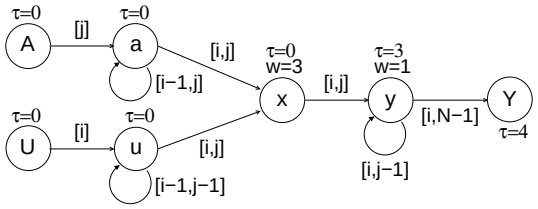

Fig. 2. Reduced dependence graph with annotated schedule for an example FIR filter algorithm.

Well known partitioning techniques are multiprojection, LSGP (local sequential global parallel, often also referred as clustering or blocking) and LPGS (local parallel global sequential, also referred as tiling). Partitioning is employed in order to match an algorithm to given architectural constraints in functional resources, memory and I/O bandwidth. Hierarchical partitioning methods apply multiple hierarchies of tiling [2].

Allocation and Scheduling. An important step in our design flow is the *space-time mapping* which assigns each iteration point $I \in \mathcal{I}$ a processor index $p \in \mathcal{P}$ (allocation) and a time index $t \in \mathcal{T}$ (scheduling) as given by the following affine transformation:

$$\begin{pmatrix} p \\ t \end{pmatrix} = \begin{pmatrix} Q \\ \lambda \end{pmatrix} \cdot I + \begin{pmatrix} q \\ \gamma \end{pmatrix} \tag{1}$$

with $I \in \mathcal{I}$, $Q \in \mathbb{Z}^{s \times n}$, $\lambda \in \mathbb{Z}^{1 \times n}$, $q \in \mathbb{Z}^s$, and $\gamma \in \mathbb{Z}$. $\mathcal{T} \subset \mathbb{Z}$ is called *time space*, that is the set of all time steps where an execution takes place. $\mathcal{P} \subset \mathbb{Z}^s$ is called *processor space*. Q is called *allocation matrix* and determines the processor that executes an iteration point I. λ is called *schedule vector* and provides the start time of each iteration point I.

Having hardware implementation in mind, it is not enough to assign a start time to each iteration point. In lieu thereof, one must determine the start time of each operation in the loop body. Therefore, we extend the time mapping in order to include the offset for the computation of each left hand side variable in the loop body and each node in the RDG, respectively. For node v_i, let this offset be $\tau(v_i)$. The overall start time of node v_i at iteration point I is: $t(v_i(I)) = \lambda \cdot I + \gamma + \tau(v_i)$.

The purpose of *scheduling* is to determine the optimal schedule vector λ (global scheduling), and the value of $\tau(v_i)$ for each node v_i (local scheduling). The purpose of *binding* is to assign each node a functional unit (*resource*) which can execute the node functionality. After binding is done, each node is associated an execution time which is denoted by $w(v_i)$. The problem of resource constrained scheduling and binding is solved by mixed integer linear programming (MILP) similar as in [7, 12].

3 Hardware Synthesis

This section describes the synthesis of hardware accelerators in form of a processor array. The synthesis both of the array generates a completely platform and language independent register transfer level (RTL) description of the hardware as intermediate representation. This representation is subsequently retargeted to HDL code (e.g., VHDL). The synthesis of a regular processor array consists of several steps: synthesis of processor elements, the array interconnection structure, and the control path.

Synthesis of Processor Elements. A processor element (PE) consists of the processor core and a local controller. The processor core implements the data path where

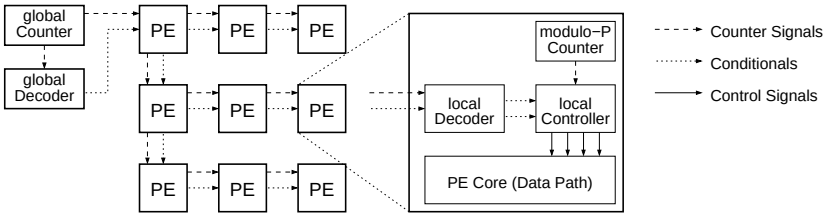

Fig. 3. Overview of control architecture. The data path interconnect is not shown.

the actual computations are performed. The synthesis of the processor requires a scheduled reduced dependence graph. During the binding phase, each operation in the loop body was assigned a functional unit (resource) that executes the operation. These functional units are instantiated in the processor core. In case of reuse of functional units, input multiplexers may be required in order to select the correct operands in every time step. The interconnection between the functional units can be directly derived from the reduced dependence graph.

Synthesis of Interconnection Structure. In order to synthesize the processor interconnection structure, one must analyze the data dependencies of the program and use the scheduling and placement information. Therefore, given the following equation of an input specification, $x_i[I] = \mathcal{F}_i(\ldots, x_j[I+d_{ji}], \ldots)$ if $\mathcal{C}_i^{\mathrm{I}}(I)$ and a space-time mapping as in Eq. (1). The synthesis of processor interconnection for the data dependency $x_j[I+d_{ji}] \longrightarrow x_i[I]$ is done by determining the *processor displacement* as $d_{ji}^p = Q \cdot (I+d_{ji}) - Q \cdot I = Q \cdot d_{ji}$. For each pair of processors p_t and $p_s = p_t + d_{ji}^p$ with $p_s \in \mathcal{P} \ \wedge \ p_t \in \mathcal{P}$, a corresponding connection is drawn from the source processor p_s to the target processor p_t. The *time displacement* denotes the number of time steps that the value of $x_j[I+d_{ji}]$ must be stored before it is used to compute $x_i[I]$. In processor arrays, the time displacement is equal to the number of delay registers on the respective processor interconnection. The time displacement is given by $d_{ji}^t = \lambda \cdot (I+d_{ji}) + \tau(x_j) + w(x_j) - \lambda \cdot I - \tau(x_i) = \lambda \cdot d_{ji} + \tau(x_j) + w(x_j) - \tau(x_i)$. Here, $\tau(x_j)$ is the offset of the calculation of x_j, and $w(x_j)$ is the execution time of the associated operation. If the number of delay registers is large or the delay registers are only filled sparsely, FIFOs, or embedded memories are selected instead.

Synthesis of Control Structure. In the context of regular processor arrays, synthesis of efficient control structures is of utmost importance for producing control signals to orchestrate the correct computation of the algorithm. The size of the control path should be as independent of the problem size and the processor array size as possible in order to ensure the high scalability of regular arrays. The key characteristic of our control methodology as depicted in Fig. 3 is the use of combined global and local control facilities. All control signals that are common for all processor elements are generated by global control units and propagated through the array, whereas local control is only necessary for signals that differ among the processor types. This strategy reduces significantly the required area and improves the clock frequency [1]. The central component of the control architecture is a global counter which generates the non-constant parts of the iteration vector. The counter signals are taken to compute the iteration dependent conditionals. Here, one can identify conditionals which are independent of the current processor index and can thus be evaluated by a global decoder unit. Only the processor dependent conditionals are subject to evaluation by per-processor local decoders.

Table 1. Experimental results.

Algorithm	No. of PEs	No. of LUTs	No. of FFs	No. of MULTs	No. of BRAMs	max. Clock (MHz)	Exec. Time (cycles)	avg. Output Interval (cycles)
Edge Detection								
– 100×100 image, partitioned	1×4	2962	1455	0	4	143	$7.7 \cdot 10^3$	3
– 1000×1000 image, partitioned	1×4	2997	1913	0	44	120	$7.5 \cdot 10^5$	3
Gaussian Filtering								
– 100×100 image, 3x3 mask	3×3	655	1439	9	2	171	$1.0 \cdot 10^4$	1
– 1000×1000 image, 3x3 mask	3×3	683	1463	9	2	171	$1.0 \cdot 10^6$	1
– 2000×2000 image, 3x3 mask	3×3	696	1472	9	4	169	$4.0 \cdot 10^6$	1
– 1000×1000 image, 5x5 mask	5×5	1538	3909	25	4	171	$1.0 \cdot 10^6$	1
FIR Filter								
– 64 Taps, partitioned	1×4	773	834	4	0	125	71	16
– 64 Taps, partitioned	1×8	1915	1014	8	0	132	71	8
– 64 Taps, projected	1×64	5782	9089	64	0	167	68	1
Matrix Multiplication								
– 6x6 matrix size, sequential	1	204	157	1	0	131	250	6
– 6x6 matrix size, partitioned	2×2	829	795	4	0	115	72	1.5
– 6x6 matrix size, projected	6×6	1888	4067	36	0	166	20	0.28
Discrete Cosine Transformation	2	1754	1152	8	1	130	94	0.65
Elliptical Wave Digital Filter	1	1169	624	1	0	94	$2.5 \cdot 10^5$	1
Partial Differential Equation Solver	1	619	502	1	0	128	$1.2 \cdot 10^5$	(1 result)
MPEG2 Quantisizer	1	637	1190	1	0	141	222	2.95
JPEG Loop 1	1	82	79	0	0	224	63	1
JPEG Loop 2	1	570	1139	0	0	158	126	1

The globally evaluated conditionals are propagated as Boolean signals along with the counter signals with the appropriate delay through the processor array. Furthermore, several operations scheduled at each iteration (with offset $\tau(v_i)$), so additional logic is required to assure the correct execution behavior. During every iteration of an iterative schedule the same sequence of control signals must be generated. Thus, the control functionality can be implemented by a modulo counter whose output is connected to a decoder logic along with globally and locally evaluated iteration dependent conditionals. This decoder generates the control signals for the functional units and multiplexers.

4 Case Studies

We have synthesized several algorithms from various application domains, some of which are taken from the well-known MediaBench suite [8]. We profiled the JPEG and MPEG2 algorithms and identified some of the most computational intensive loop kernels. The results, and the results from some more algorithms, are shown in Table 1. All algorithms were implemented using 16-bit integer or fixed point arithmetics, respectively, and synthesized using the *Xilinx ISE 6.3i* toolchain targeting a *Virtex-II 8000* FPGA. For each example, the table shows the dimension of the processor array, the cost in terms of FPGA primitives, the maximum clock frequency, the total execution time for the algorithm in clock cycles, and the average number of clock cycles between the availability of two successive output instances (for example samples, pixels). The latter is the inverse of the throughput, that is, a smaller number denotes a higher throughput. An initial latency does not affect the throughput. Note that an output interval less than 1 means that more than one output instances are available per clock cycle. The Gaussian filter was partitioned such that implementation costs remain mostly constant for larger

image sizes — of course, the latency raises. For the FIR filter, we used partitioning to trade throughput and cost at constant latency. The projected implementation is fast but very expensive, whereas partitioning allows for a fine-grained design space exploration. Similar are the results for matrix multiplication but partitioning was applied in a way that the total execution time can also be selected according to the user's requirements.

5 Conclusions

In this paper we presented PARO, a novel tool for the continuous design flow for the mapping of computationally intensive nested loop programs onto parallel processor arrays that are implemented in FPGAs. Starting point of our approach is a functional programming language which preserves the inherent parallelism of a given application. Partitioning is used as a core transformation in order to match a given algorithm to hardware constraints and user requirements. For the first time, hierarchical partitioning is supported in the whole design flow, that is, during high-level transformations, allocation, scheduling, hardware synthesis, and HDL generation phases.

References

1. H. Dutta, F. Hannig, H. Ruckdeschel, and J. Teich. Efficient Control Generation for Mapping Nested Loop Programs onto Processor Arrays. *Journal of Systems Architecture*, 53(5–6):300–309, May 2007.
2. H. Dutta, F. Hannig, and J. Teich. Hierarchical Partitioning for Piecewise Linear Algorithms. In *Proceedings of the 5th International Conference on Parallel Computing in Electrical Engineering (PARELEC)*, pages 153–160, Bialystok, Poland, Sept. 2006.
3. H. Dutta, F. Hannig, J. Teich, B. Heigl, and H. Hornegger. A Design Methodology for Hardware Acceleration of Adaptive Filter Algorithms in Image Processing. In *Proceedings of IEEE 17th International Conference on Application-specific Systems, Architectures, and Processors (ASAP)*, pages 331–337, Steamboat Springs, CO, USA, Sept. 2006.
4. Forte Design Systems. `http://www.forteds.com`.
5. A. Guillou, P. Quinton, and T. Risset. Hardware Synthesis for Multi-Dimensional Time. In *Proceedings of IEEE 14th International Conference on Application-specific Systems, Architectures, and Processors (ASAP)*, pages 40–50, The Hague, The Netherlands, June 2003.
6. S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations. In *Proceedings of the 16th International Conference on VLSI Design*, pages 461–466, Jan. 2003.
7. F. Hannig and J. Teich. Resource Constrained and Speculative Scheduling of an Algorithm Class with Run-Time Dependent Conditionals. In *Proceedings of the 15th IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP)*, pages 17–27, Galveston, TX, USA, Sept. 2004.
8. C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.
9. Mentor Graphics Corp. `http://www.mentor.com`.
10. H. Ruckdeschel, H. Dutta, F. Hannig, and J. Teich. Automatic FIR Filter Generation for FPGAs. In *Embedded Computer Systems: Architectures, Modeling, and Simulation, 5th International Workshop, SAMOS 2005*, volume 3553 of *Lecture Notes in Computer Science (LNCS)*, pages 51–61, Island of Samos, Greece, July 2005.
11. Synfora, Inc. `http://www.synfora.com`.
12. J. Teich, L. Thiele, and L. Zhang. Scheduling of Partitioned Regular Algorithms on Processor Arrays with Constrained Resources. *J. of VLSI Signal Processing*, 17(1):5–20, Sept. 1997.
13. S. Thompson. *Haskell: The Craft of Functional Programming*. Addison Wesley, 1999.