PACE: Processor Architectures for Circuit Emulation

Reiner Kolla, Oliver Springauf

Lehrstuhl für Technische Informatik, Universität Würzburg
e-mail: {kolla,springau}@informatik.uni-wuerzburg.de

Abstract. We describe a family of reconfigurable parallel architectures for logic emulation. They are supposed to be applicable like conventional FPGAs, while covering a larger range of circuit sizes and clock frequencies. In order to evaluate the performance of such programmable designs, we also need software methods for code generation from circuit descriptions. We propose a combination of scheduling and routing algorithms for embedding calculations into the target architecture.

1 Introduction

Programmable Logic Devices (PLDs) for the emulation of synchronous circuits are a big success in the area of prototyping and low volume production, especially in the form of FPGAs. These devices are user-programmable and reusable (at least the SRAM-based types). Software exists for automatic generation of configuration code from any synchronous multi-level logic design. Previous approaches to logic emulation fall into two main categories, marking the extremes of the available architectural space:

The Parallel Approach Circuits programmed into conventional FPGAs are evaluated once in every clock cycle, all their gates being processed in parallel. This approach is called *space-multiplexed computation*. FPGAs can operate at high clock frequencies, but in many cases a large part of the chip's area is left unused due to limitations in the interconnecting network[1]. Multi-cycle evaluation models have been proposed in the form of of DPGAs (dynamically programmable gate arrays, [2]) and TSFPGAs (time-switched FPGAs, [2]), where multiple CLB and switch configurations are stored on chip and a configuration identifier is distributed to all such blocks in every cycle. This leads to a better usage of chip area, since memory is the most compact representation of logic functions.

The Sequential Approach Any synchronous circuit can be evaluated by sequentially processing its gates in topological order. Thus, the simplest solution for building an emulation device would be to use a processor with a simple logical unit and sufficient memory to store the circuit's internal signal values. Such an architecture has been proposed by Jones and Lewis in [4]. The VEGA design (virtual element gate array) simulates an FPGA by evaluating one CLB at a time. We call this a *time-multiplexed computation* approach. It can be expected to be slow in comparison to the original circuit, but very compact in terms of chip area.

Between these extremes there is a large space of possible architectures for computation in both space and time. It is our hope that we can find designs that are faster

[1] As a measure for area efficiency, we use *functional density* as introduced in [2]. A design's functional density is the number of gate evaluations per area unit (λ^2) and cycle time (t_c).

than the sequential emulation devices and at the same time more area-efficient than the full-parallel, FPGA-like devices. Therefore, we will concentrate on fine-grain parallel processor designs and their applicability as PLDs. In this field, software tools for automatic code generation are an important item.

2 The PACE Architecture Space

In order to describe a processor design at a higher level, we introduce a specialized hardware description language (HDL), consisting of definitions of the basic building blocks (component types) and a set of rules for their composition – the synthesis of complex designs. In particular, this description can also serve as a target specification for the generic compiler described in the next sections.

2.1 The Component Library

First of all, we need to classify the component types that a design can be composed of, together with a behavioral description for each component. To make them available to the designer, we build a library of components and give design rules for their composition. The library shown in figure 1 contains only the most important component types, but it can easily be extended.

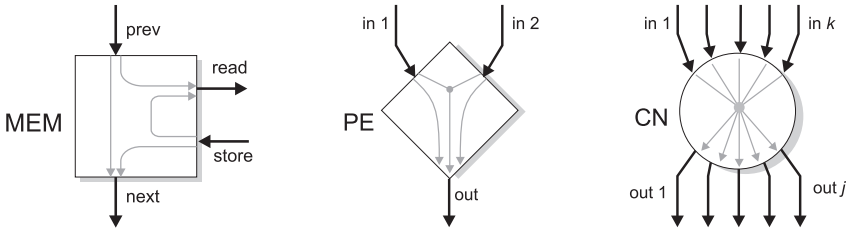

Fig. 1. Basic library components: Memory Banks (left), Processor Elements (center) and Interconnection Networks (right). Possible signal paths are shown as grey lines.

Memory Banks are used to hold the intermediate signal values during a single evaluation of a circuit. They can also be used to access the external connections of the chip by embedding I/O ports into its address space. We use single-port memory banks of any requested size.

Processor Elements (PEs) are execution units for function evaluation. The function is coded into the unit's instruction bits. Generally, a PE has one or more inputs and one or more outputs of different bit widths. For the basic library, we will use lookup-table (LUT) based PEs with one output for the evaluation of boolean functions. The number of inputs (thus the size of the LUT) can vary.

Interconnection Networks (CNs) serve as configurable communication resources. Generally, they consist of an array of switches with multiple inputs and outputs, capable of realizing different interconnections between them. The particular connection topology (crossbar, permutation, multiplexer network ...) is part of its behavioral description, i.e. for each topology there is a set of rules that describes its abilities and restrictions.

2.2 Behavioral Description

A set of rules determines the characteristic *behavior* for each component type, consisting of (a) the valid configurations (signal-to-pad assignments for inputs and outputs) and (b) the communication capabilities (the availability of signal paths *through* a component, shown as grey lines in figure 1). Component types are modelled as C++ classes in our software tool. Their behaviour is accessible via an abstract base class, making it easy to derive new component classes.

2.3 Architecture Graphs

PACE architectures are described by *architecture graphs* (AGs), composed of components (nodes) and interconnecting busses of different bit widths (directed edges). Since PACE designs are required to be synchronous and all components have zero delay, we introduce a latch symbol to model the desired behavior. Figure 2 shows the AG of a simple PACE design.

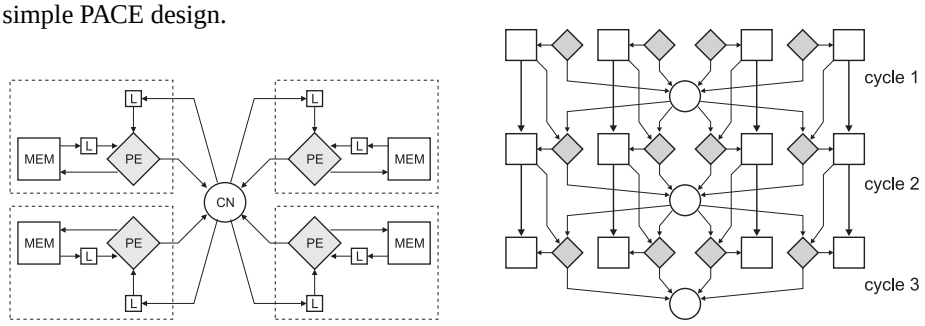

Fig. 2. Architecture graph (left) for a sample PACE design with four PEs (2-input LUTs) with local memory and its host graph (right) for three clock cycles

3 Data Flow Representation

It would be very convenient to have a common representation of calculations and data flow in the architecture over several clock cycles. Such a representation can simplify the execution planning problem. Therefore, we introduce the *host graph*, which is constructed by unrolling the AG over the time. Given an architecture graph A and a number T of clock cycles, the host graph H is built as follows:

- For any node K in A (except latches) and any cycle $0 \leq t \leq T$ there is a node (K, t) in H.
- Any edge $K_1 \rightarrow K_2$ in A (where K_1, K_2 are not latches) will be inserted in H for all cycles:
$$(K_1, t) \rightarrow (K_2, t) \qquad \forall 0 \leq t \leq T$$
- Any path between nodes K_1 and K_2 in A that consists of n latches will be inserted into H between the corresponding nodes separated by n clock cycles:
$$(K_1, t) \rightarrow (K_2, t + n) \qquad \forall 0 \leq t \leq T - n$$

– If K is a memory node in A, then there will be edges between any consecutive nodes for K in H, connecting the *prev*- and *next*-ports:

$$(K,t)_{\mathrm{next}} \rightarrow (K,t+1)_{\mathrm{prev}} \qquad \forall 0 \leq t < T$$

– There are no other nodes or edges in H.

Note, that in this representation, even memory banks are interpreted as state-less devices, serving only for communication purposes. The host graph is a transformation of a computation in space and time into a purely spatial one, forming a template for the embedding of the target circuit. It is easy to see that this can always be accomplished if there is enough memory to hold the intermediate results and if the host graph is large enough (i.e. has enough clock cycles). From an existing embedding, the processor's program can easily be extracted.

4 The Code Generation Problem

For the purpose of logic emulation, we are only interested in the circuit's visible behavior, i.e., the stable signal values at its outputs in each clock cycle. Therefore, we have the freedom to apply the whole range of existing circuit transformation methods in order to transform the circuit into an equivalent one that seems more suitable for being mapped onto the target architecture. This also includes technology mapping for the technology that is given by the available PEs' capabilities.

4.1 Task System Representation of Circuits

Signals are the carriers of information in circuits. They are characterized by their net id n and the clock cycle c they belong to, written as $S = (n, c)$. Signals are either *constant* (reset values at latches) or driven by external inputs or gate outputs. Therefore, they may depend on other signals' values – possibly in previous clock cycles (if there are one or more latches in front of an input). The basic unit of circuit evaluation is an *operation* $op[g, t]$ where the output signal o_g of gate g is evaluated for cycle t:

$$(o_g, t) = f_g((i_{g,1}, t - d_{g,1}), \cdots, (i_{g,n}, t - d_{g,n}))$$

(Here, f_g is the gate's boolean function, and $d_{g,j}$ is the number of latches between the signal driver and the j-th input of g).

These dependencies can be written as a *task graph*, whose nodes represent the operations, and edges describe their precedence relation together with time distances d. Figure 3 shows an example circuit and its task graph.

4.2 Legal Circuit Embedding

We can now define the *valid mappings* of circuits into host graphs. Let T be a task system for a given circuit, H a host graph for c_H cycles of the target architecture. An embedding of one iteration of the task system (for external clock cycle c_T) into H is legal if and only if

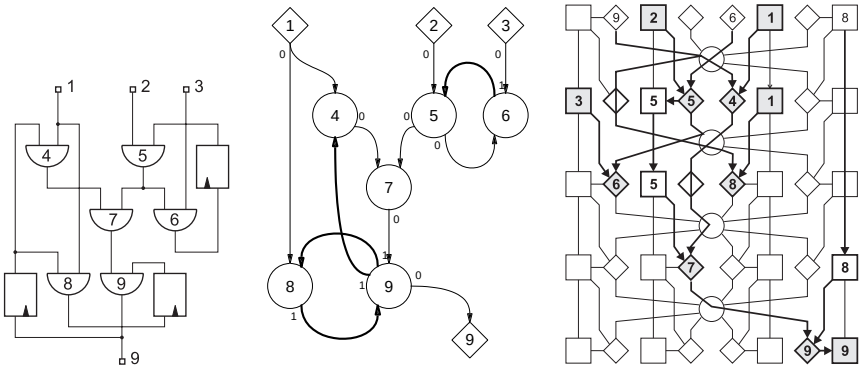

Fig. 3. A simple sequential circuit and its task graph representation (left and center). Bold edges indicate dependencies across iterations ($d = 1$). The task system was then mapped onto the architecture from figure 2 (right). Signals 6, 8, 9 (representing latch outputs) are assumed to have been calculated in previous clock cycles.

- every operation $op[n, c_T]$ (where n is a node in T) is assigned to at least one PE-node p in H, and the input and output pads of p are marked with the respective input and output signals of $op[n, c_T]$ Operations are mapped to PEs according to their precedence relation[2] in T
- for all nodes in H, the signal-to-pad assignments of their inputs and outputs meet the node's behavioral restrictions, and for all edges in H, the number of assigned signals does not exceed the edge's capacity. Data flow between processors is mapped legally onto the communication resources.

Here, we assume that previous iterations of the task system have already been embedded, so that if an operation depends on an input signal (s, t) (where $t < c_T$), this signal is either constant or it is already assigned to at least one location in H.

5 Code Generation Methods

We have implemented a generic[3] compiler tool for PACE architectures that performs both task assignment and communication planning in order to construct near-optimal emulation programs.

- A *list scheduler* is used to find valid orderings of operations, i.e. those compatible with the task systems precedence relation. The scheduler benefits from the fact that the execution of tasks from different iterations may overlap. For details on task system scheduling, see [5] and [3].
- A *placement algorithm* locates available PEs for *free* tasks (those marked by the scheduler as ready for execution)

[2] even though host graphs are not generally acyclic, they are cost-acyclic: no cycles with finite path costs exist. For this reason, precedence relations on host graphs are well-defined

[3] i.e., the target architecture's description is part of the input

– A *router* is used to map the communication (signal propagation) between the PEs onto the host graph's interconnection network. This is the only component actually depending on the architectures communication topology.
– Additional components are dealing with dealing with deadlock avoidance, signal-to-(external-)pad assignment, initialization code and program extraction.

The resulting code has the form of a loop, during which one or more evaluations of the circuit are done. At certain points, the I/O behaviour must be synchronized with the external clock signal, i.e. all output signals of the current iteration have to be processed.

Circuit	# of gates		# of 2-input LUT-PE				# of 3-input LUT-PE		
	2-LUT	3-LUT	4	8	16	32	4	8	16
C423	175	98	114 (.38)	50 (.44)	37 (.30)	36 (.15)	77 (.32)	36 (.34)	26 (.24)
C1908	314	187	195 (.40)	99 (.40)	56 (.35)	32 (.31)	135 (.35)	78 (.30)	41 (.29)
C3540	945	537	582 (.41)	295 (.40)	171 (.34)	95 (.31)	366 (.37)	179 (.38)	85 (.39)
s510	212	126	123 (.43)	67 (.40)	43 (.31)	20 (.33)	96 (.33)	42 (.38)	24 (.33)
s1488	629	373	349 (.45)	228 (.34)	103 (.38)	75 (.26)	246 (.38)	111 (.42)	58 (.40)

Table 1. Main loop length (in clock cycles) and effective processor utilization for different benchmark circuits and simple architectures. Circuits were mapped to 2-(3-)input LUT technologies using SIS. The utilization ratio is #gates / (#cycles * #PEs).

6 Summary and Results

We have presented a suggestion for new programmable logic devices, the PACE architecture family. These fine-grain parallel processor designs could bridge the gap between existing devices by providing a fast and inexpensive solution for circuit emulation. The software tools for code generation are easy to implement, since existing methods for circuit transformation, scheduling and routing can be reused and combined into a generic circuit compiler. Some of the results obtained by our experimental PACE compiler are given in table 1. The investigated architectures were variations of the design from figure 2. The results for these simple devices show an efficiency of 25%-45% and almost linear performance gain with increasing parallelism.

An extension of our concept might be the integration of other data types and processor elements, for example for doing fixed-point arithmetics. The area of application of PACE architectures might then even include typical DSP applications.

References

1. S. Brown, *FPGA Architectural Research: A Survey*, IEEE Design and Test of Computers, 9-15, Winter 1996
2. A. DeHon, *Reconfigurable Architectures for General-Purpose Computing*, A.I. Report #1586, Massachusetts Institute of Technology, October 1996
3. F. Gasperoni, U. Schwiegelshohn, J. Turek, *Cyclic Scheduling on p Processors: Optimality and Periodicity*, Report #0396, Faculty of Electrical Engineering, Universität Dortmund.
4. D. Jones, D. M. Lewis, *A Time Multiplexed FPGA Architecture for Logic Emulation*, Proc. Third ACM International Symposion on FPGAs, ACM, 1995
5. G. C. Sih, E. A. Lee, *A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures*, IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 2, February 1993