Reprint

Run-time Optimized Reconfiguration using Instruction Forecasting

M. Iliopoulos and T. Antonakopoulos

The 11th International Conference on Field Programmable Logic and Applications – FPL 2001

BELFAST, NORTHERN IRELAND, AUGUST 2001

Run-Time Optimized Reconfiguration Using Instruction Forecasting

Marios Iliopoulos and Theodore Antonakopoulos

Computers Technology Institute (CTI), Riga Fereou 61, 26221 Patras, Greece
Department of Electrical Engineering and Computers Technology,
University of Patras, 26500 Patras Greece
Tel: +30-61-997346, e-mail: antonako@ee.upatras.gr

Abstract. The extensive use of reconfigurable computing devices has imposed a new category of processors, the dynamic instruction set processors (DISPs) that customize their instruction sets dynamically to the application needs. One of the major drawbacks of DISPs is the reconfiguration time needed to alter the instruction set, which is directly added to the program execution time discouraging the use of DISPs especially for time critical processing applications. This paper introduces a methodology for optimizing reconfiguration time through instruction forecasting and presents the results obtained when applying this method to Medium Access processing systems that execute time critical network tasks.

1. Introduction

Dynamic Instruction Set Processors (DISPs) can solve the usual trade-off between performance and flexibility by tailoring their instruction sets to application needs. Runtime reconfiguration allows DISPs to implement an arbitrary long and complex instruction set by loading instructions on demand.

There is a major drawback in using Run-Time Reconfiguration (RTR). The long reconfiguration time of current devices is in some cases unacceptable, especially when time critical tasks are executed, thus there has been extensive research on methods for reducing the reconfiguration time and to expand the use of DISP systems.

There are two approaches to the problem of RTR. The first approach is related to changes in FPGA structure that can lead to reduction of configuration time, such as configuration bandwidth increase, use of partial reconfiguration or use of multiple contexts inside the FPGA [1]–[5]. The second approach is related to time optimization techniques in DISPs independently of FPGA structures, such as, exploiting temporal locality, compressing the configuration bit streams or partitioning the instructions into dynamic and static using code analysis [6]–[8].

The technique that directly reduces the reconfiguration time is the increase of configuration bandwidth. The time needed to configure a portion of reconfigurable logic is given by $T_{conf} = \frac{L}{r} + b$, where, L is the amount of data required for configuration, r is the configuration bandwidth and b is a system specific

G. Brebner and R. Woods (Eds.): FPL 2001, LNCS 2147, pp. 286-295, 2001.

configuration overhead [1]. Eventually, if we increase the configuration bandwidth the configuration time is decreased. The same effect is achieved by reducing the configuration data needed to reconfigure a portion of logic. Several devices (such as Xilinx Virtex family [2] and Atmel's AT40K family [3]) support partial reconfiguration of their resources.

The multiple context [4] or time-multiplexed FPGAs [5] are based on the idea of replicating the configuration memory within the reconfigurable device. According to this approach the multiple configurations can be stored in different internal memory planes and be selected by a global context select signal.

A general optimization technique that targets reconfiguration optimization in DISPs and is independent to the FPGA structure, is the exploitation of temporal locality [6]. According to this technique, the instructions are cached inside the DISP like in memory caches and are replaced only when there is a cache miss, otherwise the cached instruction is executed without requiring reconfiguration. Instruction caching exploits the temporal locality of instructions used in program execution. This technique is extended in [7] by partitioning the instructions into dynamic and static using code analysis, in order to achieve application specific RTR optimization. Finally, as proposed in [8], reconfiguration overhead can be reduced by applying configuration compression and thus reduce the amount of data required to reconfigure the device.

This paper proposes a new methodology for reducing the reconfiguration overhead by exploiting code analysis in DISP systems that use instruction caching, in order to pre-fetch instructions that are most likely to be used shortly. Section 2 introduces the problem of configuration bandwidth utilization in DISP processors and the basic idea of the proposed solution. Section 3 describes code analysis methodology used to extract the information needed for instruction forecasting. Section 4 outlines the scheduler implementation that exploits the instruction forecasting information while Section 5 demonstrates the experimental results obtained when applying the methodology to a MAC processing emulation system.

2. The Problem of Configuration Bandwidth Utilization

In DISP systems that use caching of instructions, the reconfiguration process is initiated each time a new instruction that has to be executed, is not contained in the cache, i.e. when there is a cache-miss. Observing the configuration bus usage in DISP systems with caching, we noticed that when there is a cache-miss, the instruction scheduler initiates a reconfiguration process that loads the missed instruction into the reconfigurable logic. When this process finishes, the scheduler remains in idle state until a new instruction miss occurs. This is illustrated in Figure 1a.

The methodology presented in this paper is based on the exploitation of the idle states of the scheduler in order to transfer an instruction that will most likely be used in the future into the reconfigurable logic (Figure 1b). This instruction is called forecast instruction. This method decreases the possibility of a potential cache miss, since an instruction that has to be executed is more likely to be into the cache due to forecasting. Although, the instruction forecasting can dramatically reduce the cache-

miss effect, it does not eliminate it, due to the fact that instruction forecasting is based on statistical analysis and occurrence probabilities.

As it is shown in Figure 1, the idea of instruction forecasting optimization is based on the fact that forecasted instructions do not stop normal program execution since there is no actual cache-miss that would put the DISP processor in a hold state. Based on this idea we introduced the code analysis methodology that produces the information for instructions forecasting using a parametric forecast window.

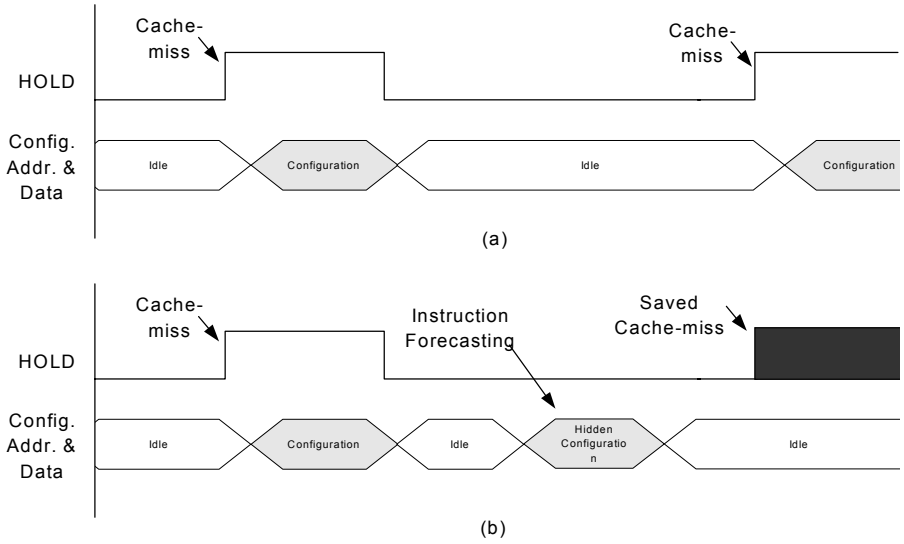

Fig. 1. (a) Normal Instruction fetching, (b) Instruction fetching with forecasting

3. Code Analysis Method for Parametric Forecasting Window

The methodology for producing the forecast instruction information is based on code analysis. More specifically, the code that has to be executed in the DISP is compiled in order to produce the assembly program. The program is parsed in order to identify the execution paths based on the branches met in the code. The parser produces dataflow graphs such as the one illustrated in Figure 2. In this DFG there are two points of interest:

- Point A, which represents the instruction that has to be executed by the DISP processor at a specific time.
- Point B, which is the instruction that is going to be executed after n instructions, where n is the length of the forecasting window. Point B is not unique and depends on the number of branches that exist between point A and point B.

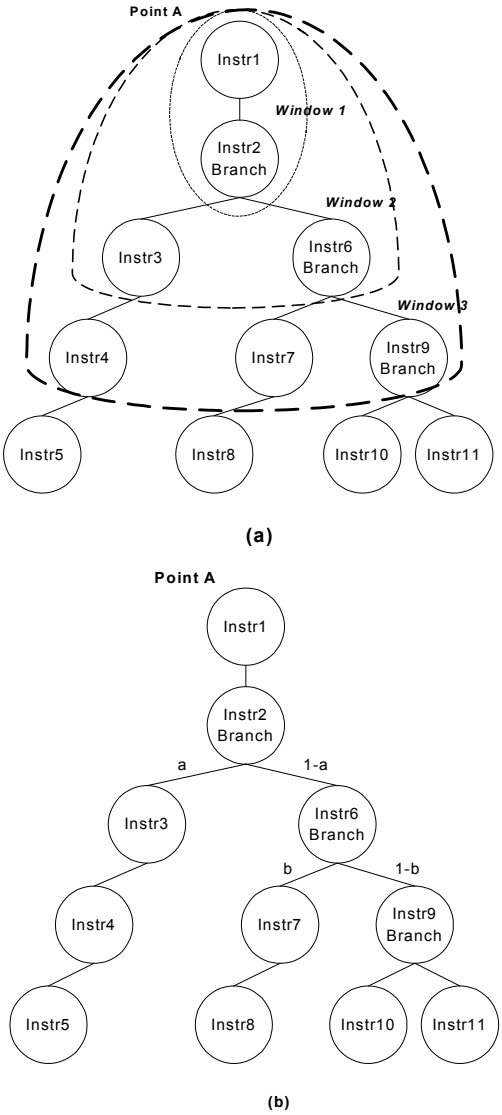

Fig. 2. Dataflow graphs (a) forecasting windows, (b) path probabilities

The instruction-forecasting algorithm uses the address space of the compiled program and initiates the forecasting field for each address. Then for each address, the algorithm checks for all possible instructions in the window with length n. For example, in Figure 2a the possible instructions in the window with length 3, starting from point A, are Instr4, Instr7 and Instr9.

In general, the probability for an instruction being executed after n instructions is the product of the probabilities of the intermediate paths. For example, in Figure 2b the forecasting for Instr1 and window length 3 is: Instr5 with probability a, Instr7 with probability $(1-a)b$ and Instr9 with probability $(1-a)(1-b)$. This probability depends on several factors such as the length of the window, the intermediate branches, and the external events that could affect the execution paths.

More specifically, the window size affects the maximum number of intermediate branches that are interleaved between point A (current time) and point B (future time). A longer-range window is more likely to achieve worse forecasting than a shorter-range window. Another factor that affects the probability of a future instruction is the kind of intermediate branches. For example, if a branch is part of a loop that is executed N times then the probability to follow the return path in the loop is N times higher than the probability to follow the path that leads outside the loop. Finally, the most important factor is the interaction of the processor with external events. In order to forecast branches that depend on external events we can emulate the system's dynamicbehavior. The results of emulation can be combined with static analysis to produce the information for forecast instructions as illustrated in Figure 3.

The outcome of the optimization method is the reconfiguration bitmaps and the reconfiguration timing information that contains the forecasting instructions used as input by the scheduler. The forecasting information is a table of addresses, each of them containing two fields, the current instruction field that is the instruction which is executed now, and the forecast instruction field, which is the instruction that is going to be executed after n instructions.

There are two approaches for integrating the forecasting information into the instruction set. The first approach embodies this information into the opcode of each instruction in order to be decoded from the instruction-decoding unit of the processor. The second approach extends the memory bus in order to directly pass forecasting instruction information to the scheduler. The second approach has the disadvantage of wasting memory for storing forecast instruction information, but since this approach is simpler, it is the one used in the presented system.

4. Scheduler Implementation for Instruction Forecasting

The RTR optimization method is implemented by a scheduler that exploits the forecast information produced by the code analysis. When a scheduler is used in common DISP systems with instruction caching, it executes the following procedures:

1. It checks the opcode of the decoded instruction and if the instruction is static or if it is already in the reconfigurable array (by checking the instruction cache), then the scheduler remains in idle state waiting for the next decoded opcode, otherwise proceeds to the next step.
2. If the instruction is not contained in the reconfigurable array, the scheduler stops normal execution flow by issuing a HOLD signal to the processor. Then the scheduler checks if there are available reconfigurable resources. If there are

available resources, it loads the instruction and returns to idle state, otherwise it replaces an existing instruction according to a replacement algorithm (such as random, First-in-First-out, Least Recently Used - LRU). Loading a new instruction or replacing an existing one causes changes into the cache and modifies the system resources. When reconfiguration is completed, the scheduler deasserts the HOLD signal allowing normal code execution.

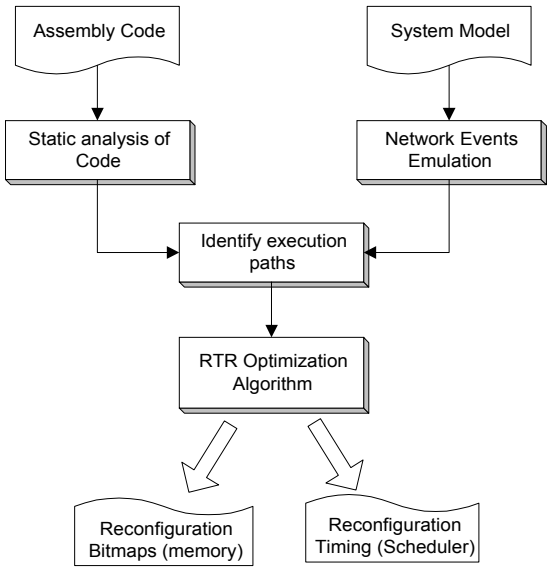

Fig. 3. Optimization of reconfiguration time

In order to exploit the forecast instruction information, the scheduler state machine has to be changed to the one shown in Figure 4. The new state machine uses the scheduler idle states for executing the following steps:

1. Each time the scheduler is in the idle state, which means that either the executed instruction is in the cache or is a static instruction, it looks for the next forecast instruction that corresponds to the current execution position. If the forecast instruction is in the instruction cache, then the scheduler rearranges the cache so that the instruction would not be replaced until it is executed (after n instructions). The rearrangement is done by using a LRU-like algorithm, i.e. the instruction that corresponds to the cache-hit position comes to the first position of the instruction queue.
2. If the forecast instruction is not in the cache, then the scheduler initiates a fetching cycle of the instruction by replacing the instruction that is at the end of the instruction queue. The whole process is transparent to the processor execution flow until there is a cache miss. In that case, the processor is in hold state until the forecast instruction is loaded.

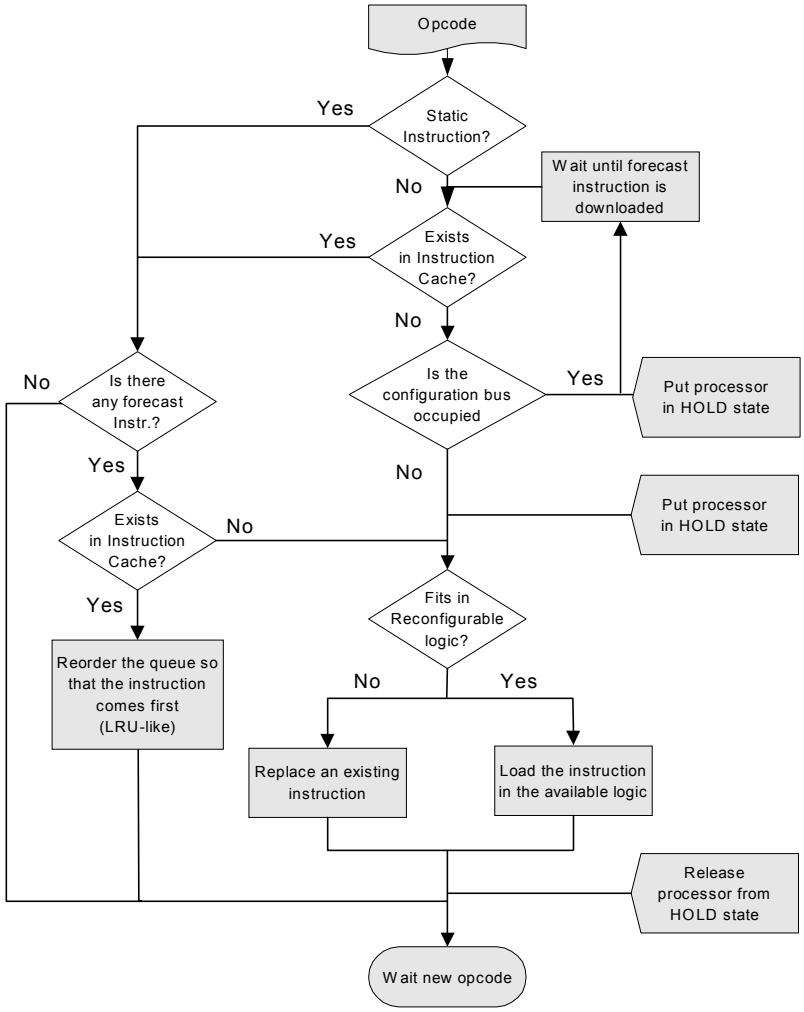

Fig. 4. State diagram of the scheduler that exploits forecast instructions

5. Experimental Results

In order to evaluate the effectiveness of the method we emulated a dynamic version of the ARM processor and we constructed an emulation platform, which is illustrated in Figure 5. The dynamic ARM processor contains the same set of instructions with the static version but part of the instruction set is executed dynamically using the scheduler described in the previous section. Two MAC processing systems were tested using the emulation platform, the IEEE802.11 [9] and the IEEE802.3 MAC

protocol [10] with the addition of the scheduler and a monitor module, which traced exchanged signals and recorded statistical information. The monitor module recognized instructions and different scheduler modes (cache-miss, cache-hit, static-instruction) and gathered statistical information for the size and the content of the instruction cache, the forecast instruction performance, like the number of cache-misses, the number of cycles that the processor is in HOLD state, etc.

The emulation platform contained also all the procedures that emulated real network events. For example, there were procedures for transmitting and receiving network packets for both protocols, procedures for host transaction emulation, like uploading or downloading data from the packet memory etc. Finally, there was the MAC implementation software, which was executed by the ARM and was used for the forecast instructions analysis.

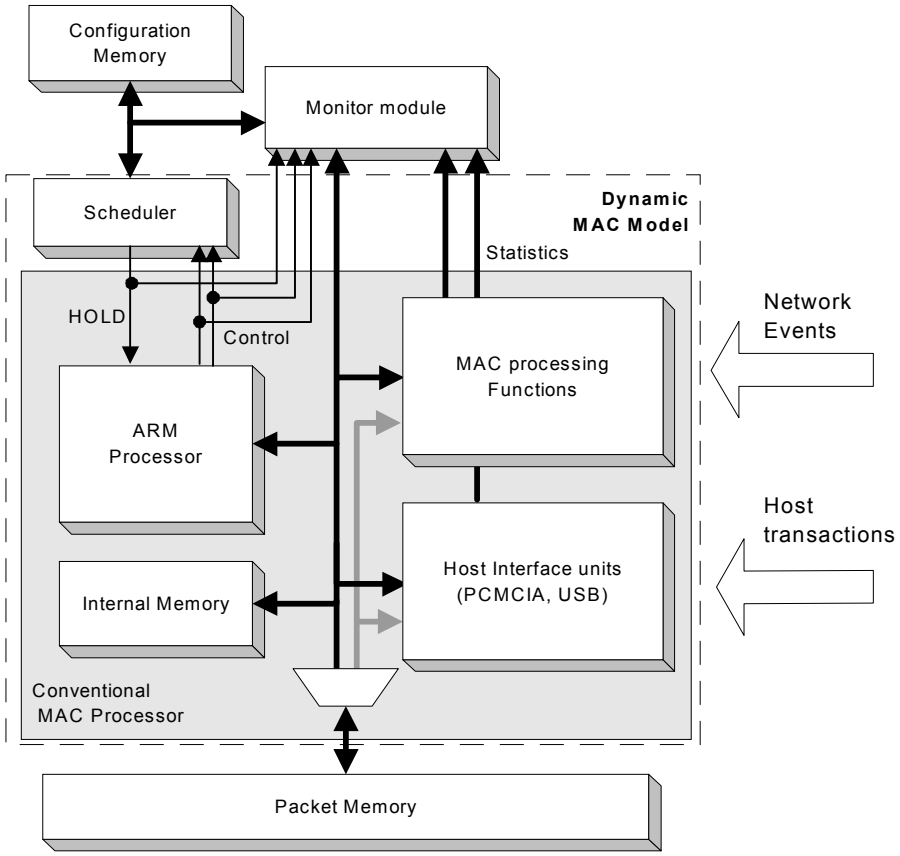

Fig. 5. Emulation platform for Dynamic MAC processing systems

The dynamic version of the MAC processing program was executed for 11 different partitions of dynamic and static instructions keeping the system resources constant (i.e. Half of the cache resources were used to implement the instructions statically). Five (5) different ranges for window forecasting were used. Each range was calculated in instruction downloading time units. The instruction downloading ranged from 2 to 12 time units, while the window sizes ranged from 8 to 20. Figure 6 shows the experimental results for different forecasting windows along with simple (no instruction forecasting) FIFO and LRU replacement algorithms. We observe that forecasting algorithm achieved an improvement in the range of 30 to 50% in cache-misses (and thus reconfiguration time). All forecasting window sizes have better performance than the simple algorithms. In our experiments the best forecasting window is equal to 16.

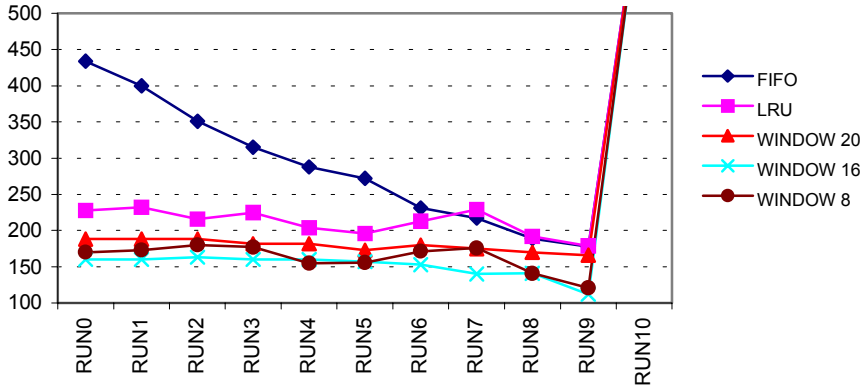

Fig. 6. Cache-misses for FIFO, LRU and window forecasting algorithm

6. Conclusions

In this paper we introduced a new method that decreases the reconfiguration time in DISP systems with instruction caching independently of the FPGA architecture or instruction swap technology and uses instruction forecasting. We demonstrated the results obtained in a MAC processing emulation platform by analyzing the IEEE802.11 and IEEE802.3 MAC protocols. The new system flow that contains the method described in this paper uses the following steps: The assembly program generated by the compiler is analyzed by an optimizer in order to generate an instruction mix of dynamic and static instructions. The instruction mix is further analyzed by the reconfiguration optimizer to produce an extended instruction set that contains forecast instruction information for the scheduler.

References

1. Michael J. Wirthlin, Improving Functional Density Through Run-Time Circuit Reconfiguration, Ph.D. thesis, 1997.
2. Xilinx, Application Note: Virtex Series Configuration Architecture User Guide, Virtex Series, XAPP151, v1.3, February 2000.
3. Atmel, AT40K05/10/20/40 FPGAs with DSP Optimized Core Cell and Distributed FreeRam, Datasheet, rev. 0896B-01/99, January 1999.
4. E. Tau, I. Eslick, D. Chen, J. Brown, and A. DeHon. A first generation DPGA implementation, Proceedings of the Third Canadian Workshop on Field-Programmable Devices, pages 138-143, May 1995.
5. Steve Trimberger, Dean Carberry, Anders Johnson, and Jennifer Wong. A time-multiplexed FPGA. In J. Arnold and K. L. Pocek, editors, Proceedings of the 5th IEEE Symposium on FPGAs for Custom Computing Machines, pages 22-28, Napa, CA, April 1997.
6. M.J. Wirthlin, and B.L. Hutchings, A Dynamic Instruction Set Computer, Proceedings of the 3rd IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), 1995, pp. 99-107.
7. Iliopoulos, M., Antonakopoulos, T., Optimized Reconfigurable MAC Processor Architecture, IEEE International Conference on Electronics and Computer Systems (ICECS), Malta, 2001.
8. S. Hauck, Z. Li, and E. J. Schwabe. Configuration Compression for the Xilinx XC6200 FPGA, Proceedings of the 6th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), April 1998.
9. Iliopoulos, M., Antonakopoulos, T., A Methodology of Implementing Medium Access Protocols Using a General Parameterized Architecture, 11th IEEE International Workshop on Rapid System Prototyping (RSP), June 2000, Paris, France
10. Iliopoulos, M., Antonakopoulos, T., Reconfigurable Network Processors based on Field Programmable System Level Integrated Circuits, 10th International Conference on Field Programmable Logic and Applications (FPL), Villach, Austria, August 2000.