A Reconfigurable Design Framework for FPGA Adaptive Computing

Ming Liu[‡†], *Zhonghai Lu*[†], *Wolfgang Kuehn*[‡], *Shuo Yang*[‡], *Axel Jantsch*[†]

‡ II. Physics Institute
Justus-Liebig-University Giessen, Germany
{ming.liu, wolfgang.kuehn, shuo.yang}@physik.uni-giessen.de

† Dept. of Electronic, Computer and Software Systems
Royal Institute of Technology, Sweden
{mingliu, zhonghai, axel}@kth.se

***Abstract*—Partial Reconfiguration (PR) offers the possibility to adaptively change part of the FPGA design without stopping the remaining system. In this paper, we present a comprehensive framework for adaptive computing, in which design key points of hardware processes, system interconnections, Operating Systems (OS), device drivers, scheduler software as well as context switching are respectively concerned in different hardware/software layers. A case study is discussed to demonstrate an example of swapping a Flash memory controller and an SRAM controller in response to diverse memory access needs. Result analysis reveals a more efficient resource utilization of 52.1% I/O pads, 86.5% LUTs and 81.3% Flip-Flops, when compared to the static design with same functionalities. A small reconfiguration overhead of context switching is measured within the range from hundreds of microseconds to milliseconds. Moreover, technical perspectives are analyzed and it is foreseen to obtain great benefits with the proposed design framework in object applications of particle physics experiments.**

***Keywords*-adaptive computing, partial reconfiguration, hardware process scheduling, hardware context switching.**

I. Introduction

Adaptive computing is the paradigm in which computation algorithms may vary and be adapted to ambient conditions during system run-time. Typically an adaptive system changes the processing behavior according to its workloads, computation interests, or other environmental situations. As a consequence, benefits including higher computation performance, lower power consumption and multitasking on limited computing resources may be obtained by dynamically changing the system architecture or adjusting important parameters. One major precondition of FPGA-based adaptive computing, is the reconfigurability of the computer systems. It denotes the capability to change customized designs by loading different configware [1]. In contrast to the static reconfigurability, a more flexible technology so-called Partial Reconfiguration (PR), is able to dynamically enable the reconfiguration process on a particular section of an FPGA design while the remaining part is still operating. The PR feature provides much convenience in adaptive computing scenarios, where basic functions are to be maintained while specific algorithms or algorithm steps are stopped and adjusted. Normally partial reconfiguration is achieved by loading a new partial bitstream into the FPGA configuration memory and overwriting the current one. Thus the reconfigurable part will change its behavior according to the newly loaded configuration.

The paper will be organized as follows: In Section II, related work is addressed with respect to adaptive computing and hardware resource management on reconfigurable devices. Based on the PR technology, a comprehensive design framework for adaptive computing is presented in Section III. Design key points in different hardware/software layers are respectively concerned in this part. In Section IV, a case study is discussed to demonstrate an example of adapting a Flash memory controller and an SRAM controller on top of the Linux OS. Targeting our object applications of particle physics experiments, we analyze the technical perspectives in Section V. Finally we conclude and propose our future work in Section VI.

II. Related Work

There exist contributions concerning adaptive computing and hardware resource management on dynamically reconfigurable devices. For instance in [2] a resource allocation model is presented for load-balancing processing of multitasks. Nevertheless the complicated hardware architecture consisting of hierarchical Upper Management Units (UMU), Management Units (MU), Processing Units (PU) and Reordering Units (RU), makes it difficult and impractical for implementation. In [3], the single processor scheduling algorithm is investigated and applied to task hardware module reconfigurations. The proposed Earliest Due Date (EDD) model for synchronous tasks and the Earliest Deadline First (EDF) model for asynchronous tasks can improve the module response time when multiple designs are being alternatively loaded into multiple reconfigurable slots. Additional scheduling mechanisms and task management studies can be found in [4], [5] and [6]. However, most of the above cited investigations concentrate only on the modeling level and do not take into account practical constraints of reconfigurable designs. In [7], a practical hardware/software environment is implemented and tested to manage hardware processes (reconfigurable modules) on FPGAs, using a modified Linux kernel called BORPH. Their main contribution is to enhance the OS kernel to support hardware processes and schedule them altogether with normal software processes. Nevertheless the modification work in the OS kernel space is error prone and makes the reconfigurable platform dependent on the customized OS. It generates many difficulties to port the schedulable system for different computing scenarios. In this paper, we will present a practical and systematic

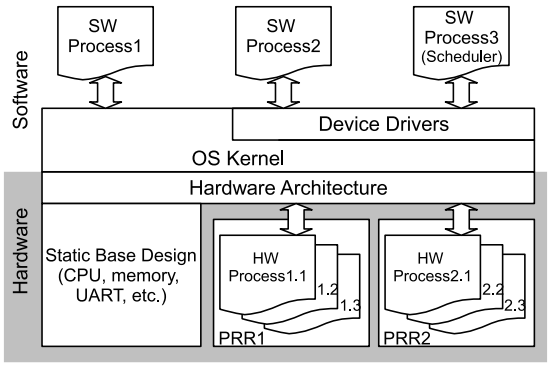

Figure 1. Hardware/software layers of the adaptive reconfigurable system

framework for reconfigurable adaptive computing, covering design issues in both hardware and software layers.

III. PR Framework for Adaptive Computing

In FPGA-based adaptive computing, different algorithm modules are individually designed according to different computation requirements. Analogous to software processes running on top of OSes, each algorithm instance can be treated as a hardware process [7] which is loaded into the PR region and runs on the FPGA fabric rather than a general-purpose processor. All hardware processes multiplex the FPGA resources in the PR region, and are scheduled to start according to certain types of disciplines on environmental conditions. Context switching happens when the current hardware process of an algorithm is being overwritten and a new algorithm is to be loaded to work. All these key issues in the adaptive computing framework are classified into and addressed within certain layers in hardware or software. Figure 1 demonstrates the layered hardware/software architecture and details in different aspects are presented in the following sub-sections respectively.

A. Hardware Infrastructure

We investigate adaptive designs using Xilinx Virtex-4 FX FPGA. The framework can also be extended on other dynamically reconfigurable FPGAs. As shown in Figure 2, the fundamental computer architecture is static, consisting of the on-chip processor (PowerPC 405 or Microblaze), Multi-Port Memory Controller (MPMC), and other peripherals on the system PLB bus. Different computation or acceleration requirements may be achieved by connecting application-specific algorithm processors, which are reconfigurable during system run-time. To be dynamically loaded with different designs, PR Regions (PRR) are reserved in the system. In the figure we show only one to demonstrate the principle. Since communications exist between PR Modules (PRM) and the static base design, specifically PLB, MPMC and I/O buffers to external channels, Bus Macros (BM) must be inserted to straddle the PR region and the base design to lock the implementation routing between them [8][9].

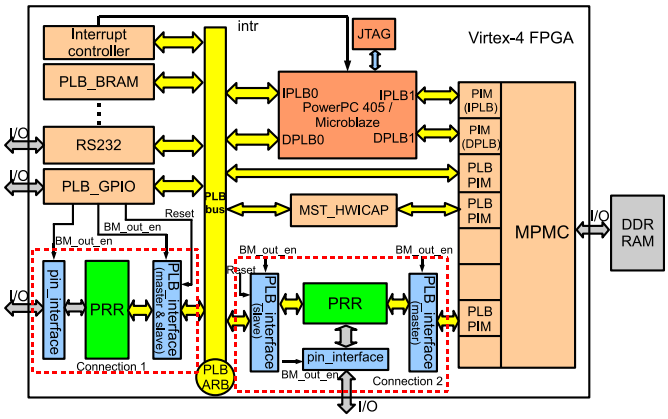

Figure 2. The PR infrastructure on Virtex-4 FX FPGA

Therefore *PLB_interface* and *pin_interface* in which BM collections are instantiated bridge all signals between PRMs and the static design. Considering the output signals from a PRM may toggle unpredictably during active reconfiguration, enable ports for BM outputs (BM_out_en) are required to disable PRM outputs and isolate unpredictable signals for the static design from being interfered. Furthermore, a separate reset signal is imported to solely reset the newly loaded module after the partial reconfiguration. It is ORed with the system PLB slave reset (SPLB_RST) and the outcoming is the actual reset applying to PRMs. Both the BM output enable and the separate reset ports are driven by a General-Purpose I/O (GPIO) core.

One significant advantage of the BM interface designs, is that they do not change the communication protocols (PLB or device-specific I/O protocols). Hence normal algorithm cores which are supposed to be used in non-PR designs, can be easily fitted in the PR region and communicate with the base design without any modification effort.

The PLB protocol consists of both the master and the slave interface. PR modules can either be entirely connected on the system PLB bus (shown in Figure 2 as connection style 1), or interface the master device directly to one port of MPMC (connection style 2) for more efficient data movement. In the latter case, the slave interface is still on the system PLB to receive controls from the processor.

B. Runtime Reconfiguration Technical Support

Run-time partial reconfiguration is the process to write partial bitstreams into the FPGA configuration memory. On Xilinx dynamically reconfigurable FPGAs, an Internal Configuration Access Port (ICAP) primitive is integrated on-chip to access the reconfiguration memory. The ICAP design in which the ICAP primitive is instantiated, interfaces to the system interconnection and transports bitstream data from memory devices (e.g. DDR in Figure 2) to load PR modules. In our previous work of [10], there are detailed descriptions on different ICAP designs with regard to their architectural

analysis and performance comparison. We choose the most practical module MST_HWICAP with a high reconfiguration throughput of 235 MB/s. Thus a typical design of hundred KiloBytes requires hundreds of microseconds (μs) for reconfiguration, which are comparable to the normal context switching overhead of software processes running on CPUs with OSes [11]. The reconfiguration time is linearly proportional to the sizes of partial bitstreams.

C. OS and Device Drivers

All hardware modules in the system infrastructure can be managed by the processor with or without OS support. In a standalone design without OS, the processor controls reconfigurable algorithm modules with low-level register accesses in application programs. While in OSes, device drivers are to be customized respectively. In a Unix-like OS, common file operations are programmed to access devices, including "*open*", "*close*", "*read*", "*write*", "*ioctl*", etc.. Interrupt handlers should also be implemented if there is the interrupt function in the hardware design. Drivers are either entirely incorporated in the OS kernel when compiled, or built into modules which can be inserted or removed according to the presence of hardware devices.

D. Scheduler Software

The scheduler of hardware processes is implemented as application programs with or without OS support. It detects ambient events and decides which algorithm to be configured next. All hardware processes are preemptable and they must comply with the management from the scheduler. Unlike the kernel space scheduling in [7] and the management unit design in hardware in [2], we manage hardware processes entirely with user space software. Major advantages include convenient portability to other platforms, avoidance of OS kernel modification, and flexibility to optimize scheduling disciplines. For example in Figure 3, a simplified scheduler program in C is demonstrated to reduce the context switching frequency and efficiently utilize computing resources. It is well suited to throughput-aware applications such as stream processing. While in realtime applications, scheduling and context switching must happen immediately, configuring the needed hardware process to respond as soon as possible. A simplified realtime scheduler example is shown in Figure 4.

E. Contextless Switching and Context Saving/Restoring

The context of hardware processes refers to buffered incoming raw data, intermediate calculation results and control variables in registers or on-chip memory blocks in PR regions. In some streaming calculations, it becomes contextless when the buffered raw data are completely digested and no intermediate state is needed to be recorded. Thus the scheduler may simply swap out the active process, and a reset will be sufficient to re-initialize it for successive

```
int scheduling(void) {
  if((data_in_fifo0 – data_in_fifo1) > THRESHOLD) {
    switching_to_hw_process = 0; // Context switching to hw process 0,
                                 // due to much buffered data in fifo0.
  }
  else if((data_in_fifo1 – data_in_fifo0) > THRESHOLD) {
    switching_to_hw_process = 1; // Context switching to hw process 1,
                                 // due to much buffered data in fifo1.
  }
  else {
    switching_to_hw_process = switching_to_hw_process; // Keep unchanging,
                                                       // to reduce reconfiguration overhead.
  }
  return switching_to_hw_process;
}
```

Figure 3. Simplified scheduler example in C for stream processing

```
int scheduling(void) {
  if(event_in_fifo0 != 0) {
    switching_to_hw_process = 0; // Context switching to hw process 0 at once,
                                 // since algorithm 0 has higher priority.
  }
  else if(event_in_fifo1 != 0) {
    switching_to_hw_process = 1; // Context switching to hw process 1.
                                 // It has lower priority, but also RT requirement.
  }
  else {
    switching_to_hw_process = switching_to_hw_process; // No event happened.
                                                       // Keep unchanging,
  }
  return switching_to_hw_process;
}
```

Figure 4. Simplified scheduler example in C for realtime applications

work when it resumes. By contrast when needed, the context saving and restoring mechanisms are very specific to concrete designs. Generally speaking, two approaches can be used to address this issue: In case of small amounts of intermediate results, register readout and writein can efficiently save the context in external memories and restore it when the corresponding hardware module resumes; When there are large quantities of data buffered in on-chip memory blocks, the ICAP interface can be employed to readout the bitstream and extract the storage values for context saving. Discussions on context saving and restoring can be referred to in [12] and [13].

IV. A Case Study with OS Support

We detail a case study in this section to verify our proposed reconfigurable framework for adaptive uses. Instead of real algorithms for specific applications, we multiplex one asynchronous NOR Flash controller and one synchronous SRAM controller in a PR region. They share the FPGA programmable resources as well as the same set of I/O buses to external devices. These two peripheral controllers are adaptively switched according to the system requirement of accessing the SRAM or the Flash memory. There exist reasons why we do not start the study from real application algorithms: 1. Our application-specific algorithms are complicated and prone to be erroneous in the hardware design. By contrast the SRAM and the Flash controllers are commercial cores and have been proved to be reliable; 2. No matter whether we use real algorithm processors or peripheral controllers for investigation, they feature the same connection interfaces to the base design (PLB) and external

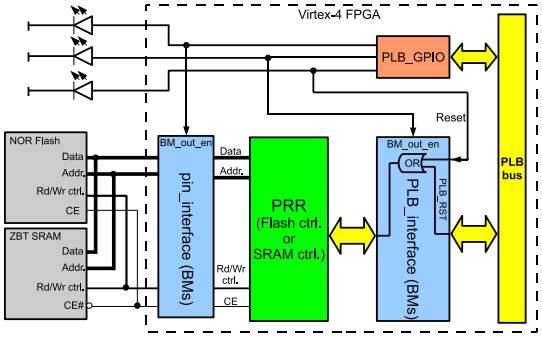

Figure 5. Flash/SRAM PR design structure

channels (IOs); 3. Multiplexing two device controllers has also practical benefits in the application design, alleviating heavy constraints from the I/O pin and other programmable resource consumption on the FPGA. In summary, it is reasonable to start from the simple and understandable memory controllers in framework verification for further adapting real algorithms.

In our FPGA design, the NOR Flash memory stores the embedded Linux kernel. It is loaded by a bootloader program into the DDR memory for CPU execution after the system power-on and the FPGA full configuration. From then on, the Flash memory will be rarely accessed only when there are external commands of upgrading the OS kernel online. For the rest of time, the IO bandwidth and the on-chip resources can be reutilized by the SRAM controller, which takes over the PR region by replacing the initial Flash controller. The SRAM targets mainly the Look-Up Table (LUT) storage usage for application-specific computation. Thus the PR region is expected to be adaptively loaded with different controller cores to react on external events, such as success of starting the OS, needs to get LUT data, or commands to upgrade the OS kernel.

A. The Flash/SRAM Reconfigurable Design

The system infrastructure on the Virtex-4 FPGA is referred to in Figure 2. The Flash/SRAM PR design is focalized in the block diagram of Figure 5. On the left side, we observe that an off-chip NOR Flash memory and a ZBT (Zero Bus Turnaround) SRAM share the same data, address and control bus I/O pins of the FPGA. These two chips are exclusively enabled by the "CE" pin. *PLB_interface* and *pin_interface* do not change their respective interface protocols. Hence the normal design of the Flash/SRAM controller can be directly implemented for reconfiguration. Since two controllers are both slave devices on the bus, master interface signals in *PLB_interface* are kept floating in fact. The BM output enable ports as well as the external reset for PR modules are controlled by a GPIO core on PLB. During the reconfiguration process, the system processor schedules the disable signals to isolate the PR region from the static design, configures the PR module, resets the newly loaded module and then re-enables BM outputs to recover the channels between PR modules and the static design. Dynamic reconfiguration work is realized by MST_HWICAP [10] loading partial bitstreams from DDR to the FPGA configuration memory during system run-time.

B. Linux OS and Device Drivers

An open-source Linux 2.6 kernel runs on the PowerPC 405 embedded platform as shown in Figure 2. To manage reconfigurable operations in Linux, device drivers for hardware IP cores have been customized to provide programming interfaces to application programs. There are respective ones for the Flash memory, the LUT block in SRAM, PLB_GPIO and MST_HWICAP. With drivers loaded, all device nodes will show up in the "/dev" directory of the Linux file system, and can be accessed by pre-defined file operations. Drivers are built into modules. They will be inserted into the kernel when the corresponding hardware IP is configured, or removed when not needed any longer.

C. Adaptive Context Switching

The scheduler program is implemented in C. Figure 6 shows the flow chart, in which the Flash memory and the SRAM controller are alternately loaded without stopping the running Linux OS and the remaining system. In this figure, steps labeled with "a - g" are used to dynamically reconfigure the SRAM controller, and the ones labeled with "A - G" are to load the Flash memory controller. Events pointed by the symbol "$\ll$" are detected by the scheduler to trigger hardware context switching. Main module switching steps before device operations include:

1) To save the register context of the to-be-unloaded module in DDR variables;
2) To remove the driver module before unloading the current hardware IP. This step is absent during the first-time system booting;
3) To disable the bus macro outputs for isolating the PR region from the base design;
4) To dynamically load the partial bitstream of the expected controller by writing its storage base address and the size to the ICAP design;
5) To reset the newly loaded controller and recover its register context;
6) To re-enable the bus macro outputs, restoring the communication between the PR region and the base design;
7) To insert the corresponding device driver module, for the processor access with high-level application software.

D. Benefit and Overhead Analysis

Multitasking features the benefit of an efficient utilization of computing resources. One obvious advantage of multiplexing IP cores in the case study, is the reduced FPGA I/O

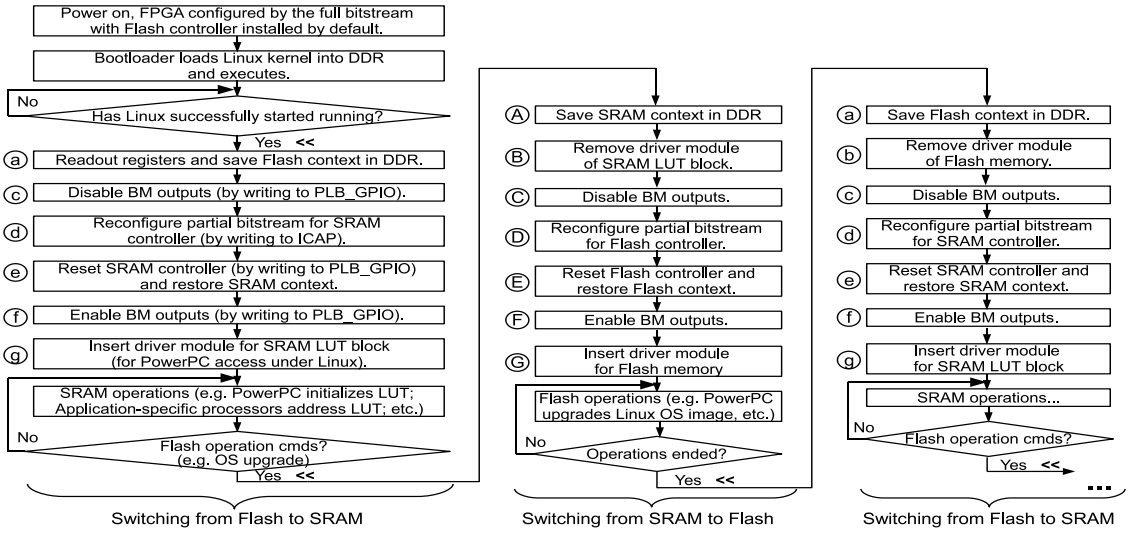

Figure 6. Flow chart of multiplexing Flash/SRAM in Linux

Resources on Virtex4-FX20	**Static Flash controller (L_1, F_1)**	**Static SRAM controller (L_2, F_2)**	**BM (L_{BM})**	**Reserved resources in PRR (L_{PRR}, F_{PRR})**	**PR resource utilization (L_{PR}, F_{PR})**	**PR vs. static resource consumption ($\frac{L_{PR}}{L_{static}}$, $\frac{F_{PR}}{F_{static}}$)**
4-input LUTs (in total 17088)	923 (5.4%)	954 (5.6%)	656 (3.8%)	1296 (7.6%)	1624 (9.5%)	$\frac{1624}{923+954} = 86.5\%$
Slice Flip-Flops (in total 17088)	867 (5.1%)	728 (4.3%)	0	1296 (7.6%)	1296 (7.6%)	$\frac{1296}{867+728} = 81.3\%$

Table I
RESOURCE UTILIZATION OF THE STATIC/RECONFIGURABLE FLASH/SRAM DESIGNS

pin count for peripheral connections, providing the possibility to connect more devices on a small FPGA with limited I/Os. Assuming a static design with both devices, the Flash memory needs 56 I/O pins and the SRAM needs 61 for data bus, address bus and control signals. In the reconfigurable design, both controllers share 61 pins. It implies only 52.1% I/O consumption compared to the ordinary non-PR design and the saving of 17.5% out of the total 320 I/O pads on Virtex-4 FX20. As a consequence, the PCB layout is simplified as well.

Comparing the traditional static design with the PR technology, the LUT and the Flip-Flop utilization on FPGAs can be analyzed as follows: For static placements of *n* algorithm cores, the overall resource consumption is the sum of all modules as shown in Equation (1) and (2) with *L* for LUTs and *F* for Flip-Flops:

$$L_{static} = L_1 + L_2 + ... + L_n \quad (1)$$

$$F_{static} = F_1 + F_2 + ... + F_n \quad (2)$$

For multiple modules sharing the same PR region, the resource consumption is respectively represented as:

$$L_{PR} = L_{PRR} + \frac{1}{2}L_{BM} \quad (3)$$

$$F_{PR} = F_{PRR} \quad (4)$$

In Equation (3) and (4), L_{PRR} and F_{PRR} are the resources reserved in the PR region for run-time reconfigurable modules. Taking into account the BM implementation with LUTs [9], half of the BM LUT utilization (L_{BM}) should be added into the overall result, since BMs straddle the PR region boundary and half LUT consumption is out of the PR region. The reserved resources in the PR region can be calculated with Equation (5):

$$L_{PRR} = F_{PRR} = Max[(L_1+\frac{1}{2}L_{BM}), ...(L_n+\frac{1}{2}L_{BM}), F_1, ...F_n]+R_{margin} \quad (5)$$

L_{PRR} is equal to F_{PRR}, since there are equivalent number (both two) of LUTs and Flip-Flops in each FPGA slice. We explain Equation (5) in the way that the PR region must be regulated large enough to contain the maximum LUT (including the other half of BM LUTs inside the PR region) or Flip-Flop utilization among all reconfigurable modules. To restrict the PR region in a rectangular shape on the FPGA, a little more resources may be retained as the margin represented by R_{margin} in the equation.

In the Flash/SRAM case study, resource consumption statistics are summarized in Table I. The first two columns list the static resource consumption of both controller designs. And the next three columns are results of the PR design. The size of the PR region is derived from the calculation using Equation (5). Considering the resource utilization of the reserved PR region and BM interfaces, we conclude that the reconfigurable infrastructure uses only

86.5% LUTs and 81.3% Flip-Flops of the static design for same functionalities. Even larger resource savings can be foreseen in practical designs, in which complex and more algorithm processors multiplex the PR region and counteract the fixed overhead of inserting BMs.

The major overhead introduced by dynamic reconfiguration is the context switching time of hardware processes. One fundamental constituent is the part that ICAP uses to download partial bitstreams and dynamically reconfigures PR modules. We measured the reconfiguration time of the Flash/SRAM controller using the MST_HWICAP core [10] and obtained the result of 299 μs for the 71.3 KBytes bitstreams. Further added by the software overhead including system calls and inserting drivers, the context switching time ranges from milliseconds (with OS support) to even within hundreds of microseconds (standalone). Fast IP switching guarantees an efficient utilization of FPGA computing resources, and promotes the PR framework applicable for applications with tight realtime requirements.

V. Technical Perspectives in Applications

The discussed reconfigurable framework is expected to be adopted in large-scale computation platforms, for instance our aimed application of data acquisition (DAQ) and trigger systems in particle physics experiments [14]. The following benefits are foreseen with the adaptive architecture:

1) **Easy design management.** Traditionally different algorithm modules are allocated and incorporated in computer systems statically by designers. The offline allocation is complex and error prone, especially in massive processing systems with hundreds of FPGAs. By contrast in a self-reconfigurable platform, the system architecture for all FPGAs becomes uniform by reserving PR regions as blackboxes. According to data signatures, different algorithm processors are adaptively loaded from the design database during system run-time. Adapting algorithms on the fly not only simplifies the design management work, but also makes it flexible and accurate.
2) **Reduced FPGA size/count requirements.** By dynamically loading different algorithms, multitasking is realized within the same reconfigurable region. It alleviates the chip size or count requirements, which should otherwise be large enough to contain all processing modules in traditional static designs.
3) **More efficient utilization of computing resources.** Flexible scheduling disciplines select computation modules according to ambient conditions. FPGA resources are less frequently kept idle and more efficiently utilized for multitasking.

VI. Conclusion and Future Work

We have presented a comprehensive framework for FPGA adaptive computing using the partial reconfiguration technology. Design key points are systematically concerned in different hardware/software layers. As a case study, a Flash memory controller and an SRAM controller are dynamically adapted according to different memory access requirements. Result analysis reveals a more efficient resource utilization and a small reconfiguration overhead of context switching. Finally technical perspectives in target applications are discussed: It is foreseen to obtain great benefits with adaptively reconfigurable computing in particle physics experiments and other large-scale processing applications.

In the future work, we will develop real algorithm processors and dynamically adapt them for experimental data processing. Sophisticated hardware context saving and restoring mechanisms are also to be investigated in depth. Moreover, inter-process communications must be efficiently regulated regarding reconfigurable modules with mutual communication requirements.

Acknowledgment

This work was supported in part by BMBF under contract Nos. 06GI179 and 06GI180, and by FZ-Juelich under contract No. COSY-099 41821475.

References

[1] C. Morra, "Configware Design Space Exploration Using Rewriting Logic", *In Proc. of the International Conference on Field Programmable Logic and Applications*, pp. 1 - 2, Aug. 2006.

[2] T. Ito, K. Mishou, Y. Okuyama, and K. Kuroda, "A Hardware Resource Management System for Adaptive Computing on Dynamically Reconfigurable Devices", *In Proc. of the Japan-China Joint Workshop on Frontier of Computer Science and Technology*, pp. 196 - 202, Nov. 2006.

[3] F. Dittmann and M. Goetz, "Applying Single Processor Algorithms to Schedule Tasks on Reconfigurable Devices Respecting Reconfiguration Times", *In Proc. of the 20th International Parallel and Distributed Processing Symposium*, Apr. 2006.

[4] H. Walder and M. Platzner, "Online Scheduling for Block-partitioned Reconfigurable Devices", *In Proc. of the Design Automation and Test in Europe Conference and Exhibition*, pp. 290 - 295, Dec. 2003.

[5] H. Walder and M. Platzner, "Non-preemptive Multitasking on FPGAs: Task Placement and Footprint Transform", *In Proc. of the 2nd International Conference on Engineering of Reconfigurable systems and Architectures*, pp. 24 - 30, Jun. 2002.

[6] C. Steiger, H. Walder, and M. Platzner, "Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks", *IEEE Transactions on Computers*, pp. 1393 - 1407, Nov. 2004.

[7] H. K. So, A. Tkachenko, and R. Brodersen, "A Unified Hardware/Software Runtime Environment for FPGA-based Reconfigurable Computers using BORPH", *In Proc. of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, pp. 259 - 264, Oct. 2006.

[8] Xilinx Inc., "Early Access Partial Reconfiguration User Guide for ISE 8.1.01i", UG208 (v1.1), Mar. 2006.

[9] M. Huebner, T. Becker, and J. Becker, "Real-time LUT-Based Network Topologies for Dynamic and Partial FPGA Self-Reconfiguration", *In Proc. of the 17th Symposium on Integrated Circuits and System Design*, Apr. 2004.

[10] M. Liu, W. Kuehn, Z. Lu, and A. Jantsch, "Run-time Partial Reconfiguration Speed Investigation and Architectural Design Space Exploration", *In Proc. of the International Conference on Field Programmable Logic and Applications*, Aug. 2009.

[11] F. M. David, J. C. Carlyle, and R. H. Campbell, "Context Switch Overheads for Linux on ARM Platforms", *In Proc. of the Workshop on Experimental Computer Science*, Jun. 2007.

[12] H. Kalte and M. Porrmann, "Context Saving and Restoring for Multitasking in Reconfigurable Systems", *In Proc. of the International Conference on Field Programmable Logic and Applications*, Aug. 2005.

[13] C. Huang and P. Hsiung, "Software-controlled Dynamically Swappable Hardware Design in Partially Reconfigurable Systems", *EURASIP Journal on Embedded Systems*, Jan. 2008.

[14] M. Liu, J. Lang, et al., "ATCA-based Computation Platform for Data Acquisition and Triggering in Particle Physics Experiments", *In Proc. of the International Conference on Field Programmable Logic and Applications*, Sep. 2008.