Multitasking on FPGA Coprocessors*

H. Simmler[2], L. Levinson[1], R. Männer[2]

[1] Weizmann Institute of Science, Rehovot, Israel 76100
[2] University of Mannheim, B6, 26; 68131 Mannheim, Germany,
eMail: simmler@ti.uni-mannheim.de

Abstract. Multitasking on an FPGA-based processor is one possibility to explore the efficacy of reconfigurable computing. Conventional computers and operating systems have demonstrated the many advantages of sharing computational hardware by several tasks over time. The ability to do run-time configuration and readback of FPGAs in a coprocessor architecture allows investigating the problems of implementing realistic multitasking. This paper explores the control software required to support task switching for an application split over the host processor – coprocessor boundary as well as the requirements and features of context saving and restoring in the FPGA coprocessor context. An FPGA coprocessor designed especially to support multitasking of such applications is described.

1 Introduction

FPGAs for custom computing machines have shown remarkable speedups for several classes of algorithms in the past years. Main reasons for high speedups are deep pipelines and parallel execution of the algorithms. In the case of FPGA coprocessors like Pamette [1], microEnable [2] or VCC HOT [3] a high datarate between the host CPU bus and the coprocessor is also an important factor in achieving high speedups. The overall performance of these FPGA coprocessor systems must be measured as a combination of execution time and data transfer from and back to the host CPU memory.

One ongoing field of research is the run time reconfiguration (RTR) of FPGAs. Most RTR approaches use these coprocessor boards as a base platform, due to the high datarate needed for the FPGA control.

RTR makes use of the reconfigurability of the FPGAs. Algorithms which use more FPGA resources than available can simply be split into parts which are then executed sequentially [4]. This is one possible use of reconfiguration. Another possibility is to execute several algorithms, that do not require all FPGA resources, in parallel in one FPGA, i.e. true multitasking [5]. A hardware manager on the CPU takes over the multitasking control and the data transfers. Due to the multiple usage of the CPU-to-FPGA coprocessor connection for the parallel executed tasks, the overall performance for each task is reduced and

* Work supported by the German-Israeli Foundation for Scientific Research and Development

logic must be added to each task to arbitrate the shared resource. A similar bottleneck is the interface to external RAM or other external devices, because these externals can be used by only one task at a time. Also each FPGA task has only a fraction of the total FPGA resources for its execution.

These bottlenecks suggest adopting a simpler usage of RTR that can be compared to a batch operation [6]. Each single task is managed through a "Virtual Hardware Manager" which is connected to the task processes. This batch method provides exclusive access to any external device and the full CPU-to-FPGA communication bandwidth. This is preserved by the execution of only one task at a time. Therefore the whole performance can be achieved for each task at the price of a possibly higher latency from start to end of execution. This can happen, because the tasks are scheduled by a special scheduling policy like first-come-first-served or shortest-task-first.

The cited RTR approaches used either total or partial overlay techniques to manage the tasks. However long tasks may block execution for minutes or even hours. A task manager must thus be able to suspend the ongoing execution of one task to avoid these blocking situations. Such a preemptive multitasking environment works like modern operating systems and must be able to extract and reconstruct the status of the FPGA designs for each task swap. A task manager suspends the ongoing execution of one task and uses the released FPGA resources to execute another task [7]. This makes it possible to build a multitasking system where each running task receives a defined time slot for its execution on the FPGA coprocessor. With the overlay technique, the FPGA design of each task must simply be loaded during initialization. With multitasking, in contrast, the FPGA design state of the preempted task must be extracted and reconstructed for each task swap.

The ideas of FPGA multitasking and the proof-of-concept implementation were briefly presented in [7]. This paper shows a more detailed list of the necessary requirements and presents the latest measurements made with a Xilinx XCV400 device. Furthermore the effects for supporting multitasking are presented and a new FPGA coprocessor board architecture is outlined. The following Section 2 lists numerous requirements needed to perform task switching and describes the task switch in more detail. Section 3 provides a brief overview of the Client-Server Model that manages the execution of several tasks. Some new architectural features especially for multitasking support are shown in Section 4. Section 5 describes the current status of our project and some recent measurements followed by the conclusions in Section 6.

2 Task Switching

Essential for implementing multitasking on FPGA coprocessors is the ability to suspend the execution of an ongoing task *and* to restore a previously interrupted task.

Such a task switch can be compared to a task switch on modern CPU's [8]. All CPU registers that define the current task's state, e.g. flag registers,

control registers and code and data descriptors, are saved into a special segment when a task switch is triggered by the operating system. This is necessary to continue the process from exactly the same state when it is later re-scheduled for execution. Then the register contents of another process to be re-started are restored to the CPU registers and this process continues execution. In case of an Intel Pentium II 104 bytes have to be stored [9].

A FPGA design normally does not have a code or data descriptor like a CPU. Rather, an FPGA holds data in several registers scattered over the FPGA. For example, data pipelines keep all data words of each single pipeline stage in separate registers, so that they are available at each clock cycle. Therefore, all used registers, latches and internal memory of an FPGA design must be saved to enable later restoration of the task. This can require up to 350 kBytes for a modern FPGA device such as the Xilinx XCV1000 [10]. In addition to saving all used register bits, there are other requirements which are listed below.

2.1 Requirements

As the central element of an FPGA coprocessor, the **FPGA** itself must provide some necessary features.

First, one must be able to extract the current state of all registers and internal memories. The determination of the internal register status is done by analyzing the readback bitstream of the FPGA. Therefore the FPGA must provide a readback bitstream that includes them. Secondly, one must be able to either preset or reset all registers and memory bits when a task is restored. This is necessary to restore the FPGA task to the state prior to the task switch. A detailed description of this task extraction and task reconstruction will be given in Section 2.2.

In addition to the requirements on the FPGAs, the **coprocessor board** must provide two features: Obviously configuration and readback of the FPGA must be supported. But to be useful, this configuration and readback must be sufficiently fast to keep task switch times reasonably small. The second requirement is complete control of the clock. Stopping the clock allows freezing the task in the current state, so that the readback can get a snapshot of all register and RAM settings at that time. Moreover, the ability to do task switching of FPGA tasks not only depends on the features of FPGA and coprocessor, but also imposes requirements on the FPGA design to be executed on the FPGA. Special attention is required when the FPGA task uses more than one clock. All clocks are constrained to be integer multiples of the slowest clock and in phase. The task must be interrupted only between cycles of this slowest clock. This ensures that when the clocks are restarted in phase no clocks are lost and all relative timings are preserved.

FPGA designs must not implement latches or registers by means of combinatorial logic loops. The state of such storage elements can neither be read back nor initialized since their state is not accessible to the configuration and readback systems of the FPGA.

A problem occurs when accessing external RAM on the coprocessor board where the address and the corresponding data are transferred on different clock cycles (e.g. synchronous and pipelined RAM [11],[12]). Allowing a task switch at any time can lead to a switch right after an addressing phase of the external RAM. The restored FPGA design will then read invalid data, because the addressed data was already presented at the RAM output. This situation can be seen at the top of Figure 1. To avoid task switches in this situation additional interface logic must generate a signal indicating when it is safe to stop the clock and to switch the task. This TS_Signal is generated by the FPGA design and can be used as an input signal of the Virtex capture block[1] [10]. Additionally to enabling the capture block also the clock has to be stopped to freeze the complete task. This is necessary because the capture block only captures the design state but does not prevent it from further accessing external devices like RAM. This can be seen at the bottom of Figure 1.

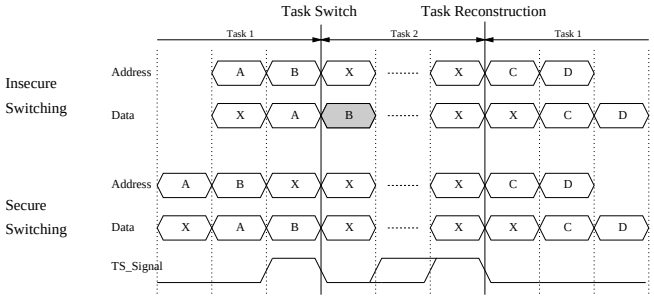

Fig. 1. External RAM access during a task switch.

Similar difficulties have to be handled when an external data source or destination is connected. A complete handshake mechanism for the data transfer is essential to guarantee proper operation. The handshake signals must be held inactive whenever the connected task is swapped out. The same logic as for the external RAM can be used here to signal the critical time when a task switch must not be made.

As mentioned before, it is essential to stop the FPGA design on a single clock and, in case of external RAM and I/O devices, only at a non-critical time.

Besides the FPGA requirements, the switchable FPGA designs and the clock control of the coprocessor boards, the **task- or hardware manager software** also imposes requirements. The task or hardware manager software must be able to extract all important state bits from the readback bitstream and must save them for their later restoration. Usually, e.g. with the Xilinx Virtex series, the readback bitstream is not suitably formatted for use as a configuration bitstream. For restoration a new download bitstream must be generated by merging the

[1] This capture block is mandatory for reading the status of the design.

extracted current state with the original configuration bit stream. The following Section 2.2 describes this state extraction and reconstruction process in detail.

2.2 Design State Extraction and Reconstruction

State extraction of a FPGA design and reconstruction of this state are the two key features to enable task switching on FPGAs. The aim of the state extraction is to ascertain all register and RAM contents that are used by the design. On the other hand, state reconstruction is to recreate the previously extracted state for each resource used in the FPGA.

State extraction of a stopped FPGA design is done by filtering all status information bits out of the readback bitstream. In order to extract these state bits, their bit positions within the readback stream must be known. Configuration information is filtered, because the logic cell configurations and their interconnections will not change at all during a task switch. The extracted state bits are then stored and form the basis for the reconstruction. Storing only the relevant information will also reduce the amount of status bits by ≈90%[2].

Task reconstruction is done by correctly setting the state of each single register or latch, and of each RAM bit in the configuration bitstream. This initialization state is normally included in the netlist file or given in an extra constraints file. Vendor specific place and route tools will then place the netlist, do all the routing and finally generate a configuration bitstream for download into the FPGA device. This whole place and route process can take, e.g., hours and is therefore unacceptable for preparing a reconstructed configuration for dynamic task switching. A direct manipulation of the configuration bitstream avoids this lengthy place and route procedure. Such a bitstream manipulation can be done for two reasons: First a task switch does not change any logic functionality or connections in the FPGA design. Secondly, the initialization information is directly coded by single bits in the configuration bitstream. All bit positions for each initialization bit in the bitstream must also be known to enable this direct and fast manipulation of the initialization states. In practice, the original bitstream is taken and each initialization bit is changed accordingly to the previously extracted register state. This is done for all used registers and RAM bits. Finally the manipulated bitstream is used to configure the FPGA. The result is then the reconstruction of the FPGA state at the moment of the task switch.

3 Client Server Model

All state extraction and bitstream manipulation must be done by a central unit in the host software system, the **Hardware Management Unit** (HMU). To

[2] For the Xilinx XCV400 device and 91% for the XCV1000.

achieve best performance the HMU must be part of the operating system itself. For a proof of concept, however, a client-server model including this HMU is sufficient and was built with only little effort for achieving good performance.

The client–server implementation was built for WinNT and uses the microEnable FPGA coprocessor [2] which includes a PCI interface, a Xilinx XC4028EX device and SRAM memory. The server is implemented as a multithreaded program that has exclusive access to the coprocessor and handles all service functions such as configure or readback. The client–server communication is bidirectional. Passing data and commands is based on interprocess communication and shared memory. A round–robin scheduling strategy was implemented within the HMU and special attention was given to the DMA tranfers to avoid blocking situations by very long DMA transfers. The measurements were done with one registered design to demonstrate the design reconstruction. Another design used external RAM to show the data consistence during swapped out FPGA design.

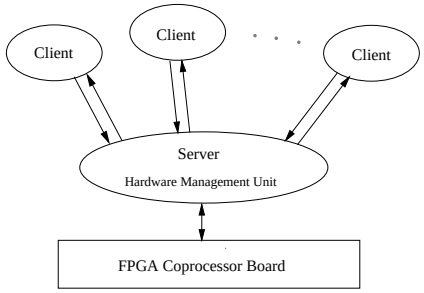

Fig. 2. Client–Server model architecture.

The possibility of task switching was successfully shown with the client–server model. However task switch efficiency was not very good because of the absence of a fast configuration and readback capability. The XC4028 was configured within ≈80 ms and readback takes ≈800 ms. The design extraction and state reconstruction for the complete FPGA was performed in 18 ms and 13 ms respectively on a 166MHz Pentium. Several important conclusions relating to configuration and to external RAM were made with this client–server model. First, the configuration/readback interface must be as fast as possible to reduce the task switch time to a minimum. Secondly, save/restore of external RAM during a task switch additonally increases the time and must be avoided.

4 An FPGA Coprocessor Designed for Multitasking

An FPGA coprocessor specifically designed to support multitasking has been outlined. The experience of the client server model and the proof-of-principle system was incorporated into the new coprocessors architecture. The coprocessor

can also be used as a modern single task FPGA coprocessor. Its most important features are:

- Fast configuration and readback capability.
  As described in Section 2.1 the time needed for a readback and for configuring the FPGA has the main influence on the system overhead during a task switch. Therefore it is essential to use a fast configuration interface.
  The new Xilinx Virtex (XCV300 through XCV800) was chosen as the FPGA device because of its eight bit parallel interface that can run at a maximum speed of 66MHz. It provides both configuration and readback. An estimated time of ≈12 ms for a task switch with configuration and readback is expected for the XCV400.
- A memory switch to avoid additional memory swapping.
  The client–server model can handle only one FPGA design at a time that has access to the external RAM. Assuming that there can be more than one design with this RAM access, the memory contents constitute part of the dynamic state that must be swapped out and back by the task switch. For several megabytes of memory, this dramatically increases the time to switch tasks. In order to avoid this additional overhead the new architecture includes a RAM switch and eight individual RAM blocks. Figure 3 shows the connection scheme of this RAM switch for several tasks. This RAM switch allows the HMU to simply disconnect RAM blocks that are assigned to a task to be suspended by a task switch. The RAM blocks of the following task to be re-activated are then connected to the same FPGA interface and the newly restored task can start execution.

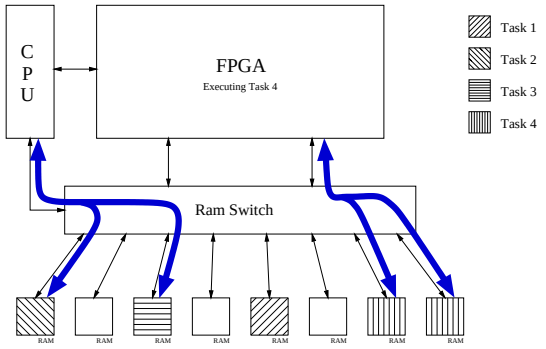

Fig. 3. RAM switch connection scheme. *The figure shows the coprocessor FPGA executing Task 4 and in parallel the HMU accessing the RAM of Tasks 2 and 3.*

- Direct RAM access during task execution.
  The RAM switch also can connect some or all RAM blocks, which are currently not used by the running task, directly to the CPU-to-coprocessor connection.

This feature enables the HMU to prepare the next task for execution by transferring its input data to its RAM block in parallel to the currently running task. This optimizes the utilization time of the FPGA. Figure 4 illustrates this in a time diagram.

– Simple data sharing between concurrent tasks.
  The mentioned relationship between a task and a RAM block can be expanded in such a way that one or more RAM blocks can be accessed by two tasks. Therefore data transfer between two concurrent tasks can be implemented easily as well as RTR with several subdesigns.

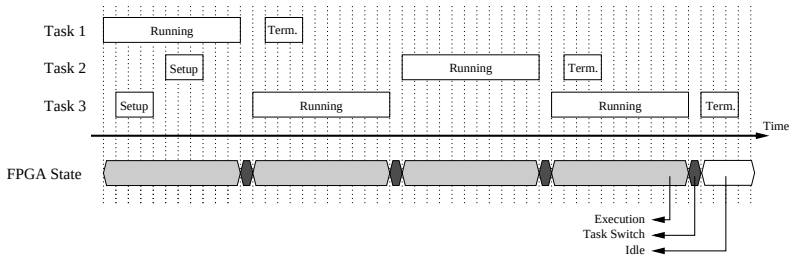

Fig. 4. Task execution timing diagram.

It must be mentioned that all running tasks can make use of the maximum performance of external FPGA interfaces like the RAM interface and the CPU-to-FPGA connection. Therefore they retain the same performance at the interfaces and only the task switch overhead has an influence on the overall performance.

The time needed to perform a task switch has to be added to the execution time for calculating the overall performance. The effect on this performance is only negligible if the task execution time is much higher than the estimated task switch time of ≈12 ms. A DES keybreak [13] or a protein structure prediction [14,15] are tasks that have such a very long execution time. Other algorithms like image processing or matrix multiplication have much shorter execution times and would require the HMU to switch the task before the time slot is over. This will result in less overall efficiency due to the task switch time but can be avoided by processing multiple data packets before a task switch occurs.

5 Current Status

The multitasking FPGA coprocessor described above is almost ready for construction.

Recent measurements done with another Virtex FPGA coprocessor board[3] have shown that a XVC400 FPGA can be configured within 12.4 ms and read-

[3] microEnableII; PCI based card with one XCV400 device.

back can be performed within 14.4 ms using the SelectedMap interface. Due to the increased amount of bits[4], the time needed for manipulating the bitstream is about 2.7 times higher on the same 166MHz Pentium. Measurements on a modern PentiumIII/600MHz have shown that only restoring the complete XCV400 bitstream can be done in ≈7.5 ms. Additional tests with the extraction/reconstruction library and real hardware were also successfull for the reconstruction of registered designs whereas the reconstruction of internal RAM will be tested in the future. Extracting the status bits from the readback bitstream was successfully shown for FPGA designs using register and internal RAM.

Concerning the HMU, the client server model has been implemented and successfully run. Some detailed planning has begun to include the manager architecture directly into the operating system. Linux has been chosen for this because it is an open system and allows the necessary modifications.

For demonstration and measurements, several algorithms are already implemented or currently under development for the new multitasking architecture. Most of them, like the protein structure prediction, have long execution times, but there are also some algorithms, such as image processing, with much shorter execution times.

6 Conclusion

This paper describes the possibility of performing multitasking of FPGA designs. The idea is to share the FPGA resources among several tasks through the use of pseudo multitasking, much as computer operating systems emulate multitasking on a single CPU.

Even though parallel execution of several tasks in the same FPGA is possible, there are several advantages for multitasking: e.g. switching between the FPGA designs of several tasks will retain the full communication data rate between the CPU and the FPGA. Although the total data rate over time is the same for parallel execution and multitasking, in the case of multitasking the I/O resources are totally dedicated to the current FPGA design. Therefore the application is not concerned with sharing these resources.

Secondly, in contrast to the overlay technique for sequential execution of several algorithm steps with different FPGA designs, the hardware manager does not need to wait until pipelines, FIFOs, etc. are completely empty. Multitasking allows switching the FPGA design at almost any time without loosing data.

The third advantage concerns programming of the FPGA design. Each FPGA design has the complete set of I/O resources available. The programmer does not need to care about resource sharing and so writing FPGA designs is much easier.

The disadvantages are almost the same as in modern multitasking operating systems. Only one task at a time is allowed to execute[5]. The completion of com-

[4] XC4028EX has ≈668 kBits; XCV400 has ≈1.75 MBits.

[5] On a single processor computer.

putations is delayed. Full efficiency cannot be achieved due to the task switching overhead.

The described client–server model was implemented to demonstrate this task switching principle for FPGAs. This model together with two FPGA designs, especially designed to check the feasibility of task switching on FPGAs, have shown that it works for the XC4000 series. Additional tests and measurements with Virtex devices have shown that it can be done within a reasonable time.

The lessons from this experience and measurements have been incorporated into the design of a new architecture, containing specific multitasking support features. Some of these important features have been described in this paper.

References

1. Mark Shand: PCI Pamette V1. DEC, Systems Research Center, Palo Alto, USA. 1997. `http://www.research.digital.com/SRC/pamette`
2. K.-H. Noffz and R. Lay: microEnable, Silicon Software GmbH, Mannheim, Germany. 1999. `http://www.silicon-software.com`
3. Virtual Computer Corporation: VCC H.O.T. II, Virtual Computer Corporation, Reseda, USA. 1997. `http://www.vcc.com`
4. R. Hudson, D. Lehn and P. Athanas: A Run-Time Reconfigurable Engine for Image Interpolation, IEEE Symposium on FPGAs for Custom Computing Machines, Los Alamitos, California. April 1998. Page 88-95.
5. G. Brebner: The Swappable Logic Unit: A Paradigm for Virtual Hardware, IEEE Symposium on FPGAs for Custom Computing Machines, Los Alamitos, California. April 1997. Pages 77–86.
6. J. Jean, K. Tomko, V. Yavagal, R. Cook and J. Shah: Dynamic Reconfiguration to Support Concurrent Applications, IEEE Symposium on FPGAs for Custom Computing Machines, Los Alamitos, California. April 1998. Pages 302–303.
7. H. Simmler, L. Levinson and R. Männer: Preemptive Multitasking on FPGAs. IEEE Symposium on FPGAs for Custom Computing Machines, Los Alamitos, California. April 2000. *unpublished*.
8. J. Nehmer and P. Sturm: Systemsoftware. dPunkt.Verlag. 1998.
9. Intel: Intel Architecture Software Developer's Manual, Volume 3, Intel Inc.. 1999. `http://www.intel.com/design/product.htm`
10. Xilinx Inc.: Virtex 2,5V Field Programmable Gate Arrays, Xilinx. San Jose, California 95124. 1999. `http://www.xilinx.com/products/virtex.htm`
11. IDT: Fast Static Rams and Modules, IDT Inc. 1999. `http://www.idt.com/products/sram/Welcome.html`
12. Samsung: SRam Products, Samsung Semiconductor Inc.. 1999 `http://www.usa.samsungsemi.com/products/browse/ntramsram.htm`
13. T. Kean and A. Duncan: DES Key Breaking, Encryption and Decryption on the XC6216, IEEE Symposium on FPGAs for Custom Computing Machines, Los Alamitos, California. April 1998. Pages 310–311.
14. H. Simmler, E. Bindewald, R. Männer: Acceleration of Protein Energy Calculation by FPGAs, Proc. Int'l Conf. on Mathematics and Engineering Techniques in Medicine and Biological Science. CSREA Press, June 2000. *unpublished*.
15. E. Bindewald, et.al.: Ab inition protein structure prediction with MOLEGO, Proc. 7th Int'l Conf. on Intelligent Systems for Molecular Biology. 1999.