Operating Systems for FPGA Based Computers and Their Memory Management

Klaus Danne, University of Paderborn, Germany

Abstract: We introduce the concept of an operating system for platforms that consist beside memory and peripheral devices of FPGAs as the only computational resource. Applications can be developed independent from each other and due to device drivers with little dependency on the platform. The OS supports the multitasking execution of applications using static as well as dynamic resource assignment. A main focus of the paper is the management of the resource memory. Memory management as part of the OS is introduced which allows multiple tasks to access the same memory banks using virtual addressing and dynamic memory allocation. Access conflicts are solved by a priority based scheduling. Since no microprocessor is part of the system, the entire OS including its memory management is executed on the FPGAs. [1]

1 Introduction and Related Work

Reconfigurable hardware devices such as field-programmable gate arrays (FPGA) and coarse grain architectures are general purpose computing devices. Consequently, we can build up computer systems which use reconfigurable hardware devices instead of microprocessors as main computing resource.

The convenient development and execution of applications for such a system requires abstraction form platform details and the management of the system resources. These two major requirements, *abstraction* and *resource management*, are usually provided by an operating system (OS).

This work introduces our concept of an OS for FPGA based computers in section 2. Several steps lead to an OS, which is able to dynamically assign platform resources to multiple applications consisting of *hardware tasks*. The second part of the paper deals with the management of the resource *memory* (section 3). We introduce our memory management unit (MMU) which allows central memory banks to be shared along several running tasks. The concepts of *virtual addressing* as well as *dynamic memory allocation* are implemented. Before this, we review some related work.

[De96] was one of the first work that investigated the computing capabilities of reconfigurable architectures and compared them to processors following the Von Neumann paradigm. The results have shown, that reconfigurable architectures are general purpose computing devices and that they can overcome Von Neumann processors in performance and efficiency for applications with certain characteristics. Many theoretical work has been done in the field of multitasking on FPGAs. [JTY$^+$99] deals with the supports of

[1]This work was partly developed in the course of the Graduate College 776 -Automatic Configuration in Open Systems- and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

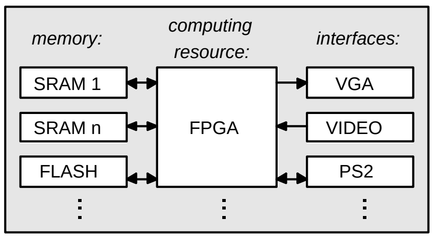


Figure 1: Architecture of FPGA Based Computer Platform

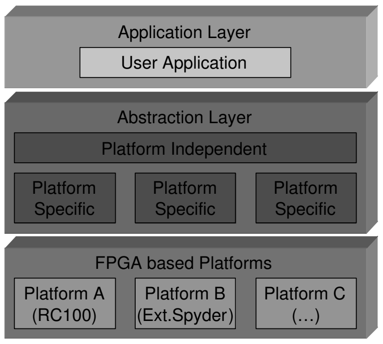


Figure 2: Abstraction Layer to Support Abstraction from Platform Details and Device Function

concurrent applications in a multi FPGA-system by reconfiguring entire FPGAs. Most of the work assuming partial reconfigurable FPGAs [LP02] is based on a model, where applications consist of a set of tasks and each task requires a certain rectangular shaped area of the FPGA. For an application specified as a directed acyclic graph of tasks with given area consumption and execution time [TFS00] introduce an optimal off-line scheduling and placement algorithm. Heuristics for on-line and off-line scheduling were analyzed in [BKS00]. The term *operating system* together with partial reconfigurable FPGA was first used in [Br96]. There, the OS supports an application running on an host processor with hardware accelerators that could be reconfigured on an FPGA. [BD01] presents a simple first fit scheduler for a one-dimensional FPGA resource model, which is implemented onto the FPGA itself. Advanced work on OS for reconfigurable devices is presented by the authors of [WP03] and [WK01].

2 FPGA Operating System Overview

2.1 Platform Overview

The task of the OS is to support the development and execution of applications on FPGA based computers. Fig. 1 gives an overview of our target architecture. The computing resource consist of one or many FPGAs. These can consist just of a pure array of logic cells and routing resources but can although include dedicated resources like block memories or multipliers on the chip. The FPGAs can either allow only the entire reconfiguration of all logic cells or allow partial reconfiguration of some logic cells while the others keep their configuration. The later method is often restricted to reconfiguration of one of many entire columns of logic cells. The FPGAs are connected to the memory resources which consist of volatile memory like e.g. multiple banks of static random access memory (SRAM) and of non-volatile memory like for instance FLASH memory. In addition, the platform has interfaces for devices of various types.

The next three subsections introduce concepts which stepwise extend the support to execute applications on a platform described above. Similar to application tasks of a conven-

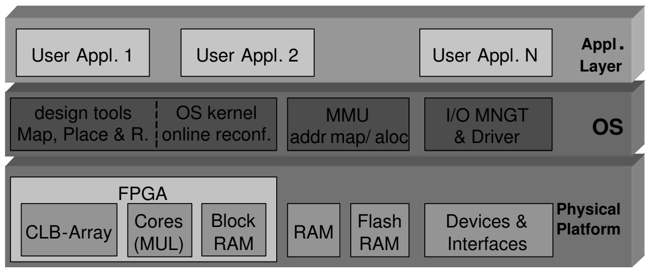


Figure 3: OS Layer for multi tasking with static or dynamic task set

tional computer system, here the applications consist of digital circuits emulated by the FPGA which are referred to as *hardware tasks* or just tasks.

2.2 Abstraction from Platform and Devices

Applications targeting such platforms are usually implemented using a hardware description language like VHDL or Verilog to define a digital circuit. The code is synthesized into an appropriate configuration bit-stream which can be loaded to the FPGA to define its behavior. If no additional support is used, the application designer has to deal with many platform details and the application code becomes much platform dependent. Moreover, the appropriate control of each device used by the application has to be implemented from scratch. Therefore the first step to facilitate the application development is the abstraction from the physical platform details as well as the abstraction from the functional details of the devices. This is done by introducing an abstraction layer between the application and the underlying hardware as shown in fig. 2. On the bottom there are the various FPGA based platforms like e.g. the RC100 system from Celoxica or an extended version of the SPYDER system from X2C. The abstraction layer consists of a part that is specific for every platform and a platform independent part. The first part includes things like the available resources and connectivity details of the particular platform (e.g. a PS2 connector is attached to the pins {x,y,z} of the first FPGA). The second part includes a set of services in form of pre-developed circuits in a library (e.g a circuit implementing a PS2 protocol stack). The application on top of the abstraction layer can now be developed mostly platform independent and can be synthesized for various target platforms.

2.3 Multi Tasking with Static Task Set

Due to the parallel nature of the FPGA, it can simultaneously execute multiple independent applications. In this case usually the code of all applications is combined and synthesized into one FPGA configuration. The synthesis tools assign separate FPGA logic cells to each applications, but the designer has to be aware that no conflict is produced by the use of all other kinds of resources (e.g. memory and devices). Fig. 3 shows, how an OS layer can manage these resources and resolve conflicts. The bottom of the figure shows the physical platform and its resources. At the top, there is the application layer with a fix set of independent user applications which should run simultaneously on the platform. The OS layer shows the mechanisms to manage the platform resources. Since

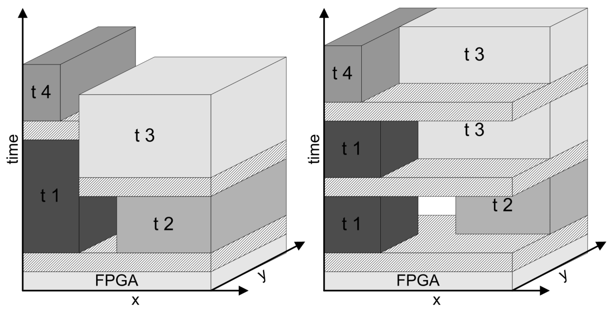


Figure 4: left: Partial reconfiguration using a column wise reconfigurable FPGA. right: Pseudo partial reconfiguration using a full reconfigurable FPGA.

the set of applications is static, the FPGA logic cells can be assigned off-line during the synthesis process by the *map & place & route* tools. In contrast to that, the off-chip memory resources have to be managed on-line. Even if each application requires a static amount of memory which can be assigned off-line, a conflict can occur when more than one application accesses memory of the same physical RAM bank in the same clock cycle. A memory management which resolves such conflicts using priority based access and furthermore allows dynamic memory allocation is introduced in the next section. A similar exclusion mechanism is required for the interfaces. Here, the device driver and the I/O management is responsible to guarantee a proper handling of the devices.

2.4 Multi Tasking with Dynamic Tasks Activation

This extension of the OS functionality allows that application tasks can enter the system and become activated[2] at runtime. Since the set of active tasks is now unpredictable, the reacquired computing resources (FPGA logic cells) of each task have to be assigned on-line. This involves a re-configuration of the FPGA during runtime. If a task becomes active the OS has to identify free FPGA resources and to re-configure them which the configuration bit-stream of the appropriate task. If there are not enough free resources on the FPGA the task cannot be *placed* and has to be *scheduled* for later execution.

In this scenario the MMU has to support dynamic memory allocation. Even if each application task requires a static amount of memory, the MMU has to be able to assign memory to a new activated task and to release it when the task terminates. The management of the devices can be extended by *on demand drivers*. The driver of a certain device, which itself is some configuration bit-stream to be executed on the FPGA, is placed on the FPGA only when an application task requests this device. Therefore the resources required by an device driver can be used by other tasks if no task is using the device.

2.5 FPGA based computer using reconfiguration

Fig. 4 illustrates two different reconfiguration techniques how tasks can be executed on an FPGA. The tasks are showed as 3D-boxes which occupy a certain x-y-rectangle of

[2]A task is set to be active if it has entered the system and is ready for execution. The activation can be result of an external event e.g. a user input or some internal event.

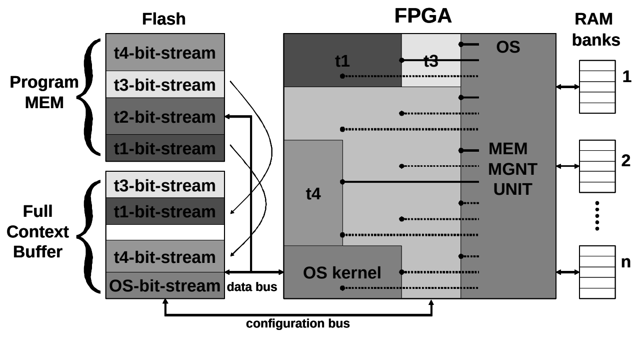


Figure 5: Architecture of an FPGA based computer. Multi tasking is done using the pseudo partial reconfiguration method. The tasks can access external memory using a MMU.

FPGA area during their execution time. The hatched boxes denote that this FPGA area is currently reconfigured.

Fig. 4 (left) shows partial reconfiguration on a column-by-column reconfigurable FPGA. These devices allow to reconfigure a partition of the logic cells during runtime while the other partitions continue their operation. The constraint is, that only entire columns of logic cells can be reconfigured together. Therefore the resources of each task span the whole y-dimension of the FPGA area. In the initial configuration task t1 and t2 are placed on the FPGA. When task t2 terminates the area is reconfigured with task t3 without disturbing the execution of task t1. Later, when t1 terminates its occupied area is reconfigured with task t4.

Fig. 4 (right) shows pseudo partial reconfiguration on a FPGA which allows only the entire reconfiguration of its logic cells. Task t1 and t2 are loaded with the initial configuration. When task t2 terminates and task t3 should be started, the system suspends all other running tasks (here t1). The entire FPGA is reconfigured with a new bit-stream that contains the configuration data of task t1 and task t3. Task t3 is started and task t1 resumes its execution. Later t1 terminates and t3 is interrupted to load a new configuration that contains task t4 and t3. Using this method, the tasks do not have to span entire columns of the FPGA but can occupy rectangular shaped resources. The disadvantage is, that all tasks are interrupted if a new task is activated. Two major technical obstacles have to be solved to implement this reconfiguration model. 1. Since it is not predictable at design time which tasks have to run on the FPGA simultaneously, the system has to be able to generate a full bit-stream as a combination of bit-streams of single tasks during runtime. 2. The state of the task has to be preserved during its interruption. This can be solved in two ways. A) The current FPGA configuration including the state of all registers of the suspended tasks is read back before the reconfiguration via the FPGA configuration interface. During the generation of the new bit-stream the state information of the tasks which have to be resumed is build in. B) A kind of suspension procedure stores all state registers of the task in an external memory. When the task should be resumed this information is read back to the all task registers before the normal execution resumes.

Fig.5 presents the architecture of an FPGA based computer using the pseudo partial reconfiguration mode. The configuration bit-streams of the tasks are stored in a Flash memory.

The initial configuration of the FPGA includes a task named OS-kernel that controls which tasks should be executed on the FPGA and in which area. If it has been decided to execute a certain task, the appropriate bit-stream is loaded from the Flash and combined with the configurations of the tasks which are currently running. This new bit-stream is stored in the *Full Context Buffer*. After that, the execution of all tasks is stopped and the entire FPGA is reconfigured from this buffer.

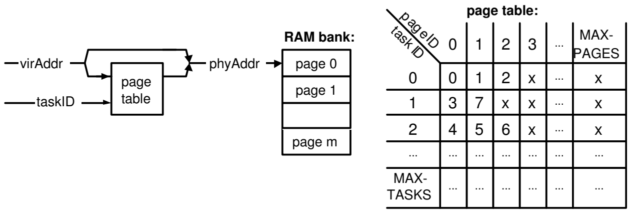
3 Memory Management

This section presents a *memory management* of an OS for FPGA based computers. Since the OS consists of emulated hardware there is no physical *memory management unit* (MMU). Here, we refer to MMU as the part of the OS which handles the memory management. Some concepts of the memory management of ordinary computer systems are adopted while other techniques are new due to the special characteristics of the system.

Like in other computer systems, the memory is a resource to be shared by several active tasks. In general the tasks are designed independently and at design time it is not known which combination of tasks will be executed together at runtime. Therefore the MMU should support *virtual addressing*.

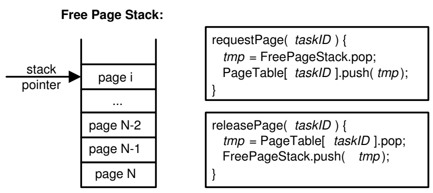
On the other hand there are several differences in the organization of this computer system which effect the memory management.

- The program memory (here configuration memory) and the data memory are physically separated. The configuration memory is distributed on the chip and can be accessed via a configuration interface. Data memory of a task can be a combination of task registers which are distributed on the FPGA and dedicated memory which can be on-chip or off-chip RAM. Therefore the MMU presented here manages only the use of dedicated data RAM.
- The presented FPGA computer architecture has no hierarchy of data memories with different size and performance. Therefore, up to now there is no need for memory *caching*.
- The FPGA computer can run multiple tasks simultaneously which can cause conflicts when more than one task wants to access the same memory bank at the same time. It is the task of the memory management to resolve this conflicts.
- In general, the architecture has multiple memory banks which can be accessed in parallel. The memory management should take advantage from this in an efficient manner.
- The tasks of the FPGA computer differ from tasks of a sequential CPU in the way, that they usually process data synchronous and in parallel. This means that the execution time of the parallel operations should be fit to guarantee a correct computation. Since multiple tasks use the same memory banks, they have either to be designed to be able to handle delays of the memory access or the system has to guarantee that each task can access the memory before some deadline.



pageID / task ID	0	1	2	3	...	MAX-PAGES
0	0	1	2	x	...	x
1	3	7	x	x	...	x
2	4	5	6	x	...	x
...	...	...	...	...	...	...
MAX-TASKS	...	...	...	...	...	...

Figure 6: Mapping the virtual address of a task to the physical address using a page table



```
requestPage( taskID ) {
   tmp = FreePageStack.pop;
   PageTable[ taskID ].push( tmp );
}
```

```
releasePage( taskID ) {
   tmp = PageTable[ taskID ].pop;
   FreePageStack.push( tmp );
}
```

Figure 7: Allocation and deallocation of memory pages

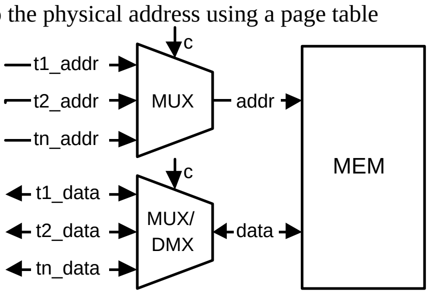


Figure 8: Multiplexed access of several tasks to one physical memory

The next subsections show how we implemented the concepts of *virtual addressing* and *dynamic memory allocation*, how the memory accesses of multiple tasks are scheduled.

3.1 Virtual Addressing

The goal is, that the data memory addressing of a task becomes independent of other tasks running in the system. Therefore each task gets a memory space assigned that begins at a virtual address 0x00…0 and ends at a certain value. If the memory requirement of each task is static and all tasks are known at design time, the virtual addresses of each tasks can be translated to physical addresses by a preprocessor tool before the synthesis happens. To do this at runtime, the MMU has to store information how to map the virtual addresses of each task to physical addresses. In our implementation the memory space of physical RAM bank is divided into fix number of pages of the same size. Every task gets memory assigned in form of one or more pages. Which pages belong to which tasks is stored by the MMU in form of a page table (fig. 6). When a task makes a memory access using a virtual address, the hi-word (upper part) of the virtual address together with the task-id is used to address the page table. The output of the page table is used as hi-word of the physical address and is equal to the base address of the page. The low-word (lower part) of the virtual address is used as the low-word of the physical address and equals the address inside the page. The width of the hi-word and low-word depends on the chosen page size. Using this simple virtual mapping mechanism the page-size has to be a power of two.

3.2 Dynamic Memory Allocation

Dynamic memory management allows task to allocate and de-allocate memory at runtime. This is useful when new tasks can enter the system at runtime or when a task itself has a dynamic memory requirement. Our MMU supports this in form of pages which can be requested or released by the tasks. Therefore, the MMU stores the set of the free memory pages and modifies the page table if tasks get new pages assigned. To be able to allocate and de-allocate a page in one clock cycle, the set of free pages as well as the set of pages assigned to each task (page table) are organized as stacks. Fig. 7 shows the stack of free pages as well as the operations of requesting and releasing a page which can be executed in one clock cycle. The *request(taskID)* function pops the pageID from the top of the free page stack and pushes it on the page stack of the particular task (the row in the page table with this taskID). After this, the task has one more page assigned to its address space and there is one page less on the free page stack. The *release(taskID)* function does the same in the opposite direction.

3.3 Priority based Scheduled Memory Access

To let multiple tasks access the same physical memory bank, the address bus and the data bus have to be multiplexed (fig. 8). By changing the control signal *c*, the MMU decides which task can access the memory. If a task requests a read or write access to the memory, the MMU switches the multiplexers to the particular task and the access can be done in the same clock cycle.

If more then one task requests an access to the memory during the same time, a scheduling of the memory accesses has to be done. To be able to guarantee timing constraints, the schedule can be done based on priorities. If multiple tasks request the memory during the same clock cycle the MMU connects the task with the highest priority. The request of the other tasks is delayed and is served, when no other task with a higher priority requests the memory.

To be able to guarantee an satisfactory schedule, the moments of memory access of each task has to be modeled. In our first approach we distinguish between tasks with periodic memory access and hard real-time constraints and tasks with aperiodic access. The time interval of an access is only one clock-cycle. The tasks with periodic access usually repeat frequently the same operation in some clock-cycles and request an memory access every P (period) cycles (fig. 9). The memory request (upward arrow) happens at the end of the clock-cycle when the address (and the data, in case of an write access) becomes ready. The deadline (downward arrow) is at the end of the clock-cycle before the data register is read or written. If a task requests the memory aperiodically without a deadline, it is designed to wait with its processing until the memory access has been done.

Fig.10 shows a schedule where tasks with periodic and aperiodic memory accesses are scheduled based on fix priority assignment. Task $t1$ accesses periodically with a period of $P = D = 3$, task $t2$ and $t3$ with $P = D = 4$. Task $t4$ accesses the memory aperiodic without a deadline. The schedule is done using a fix priority assignment, starting with task $t1$ with the highest up to task $t4$ with the lowest priority. This concept is known as rate monotonic scheduling. It can be seen in the figure, that every task meets its deadline. The aperiodic requests are served when no periodic task requests the memory.

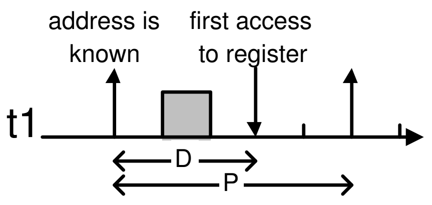


Figure 9: Model of task memory access: upward arrow = moment of request; downward arrow = moment of deadline; P = period; D = relative deadline

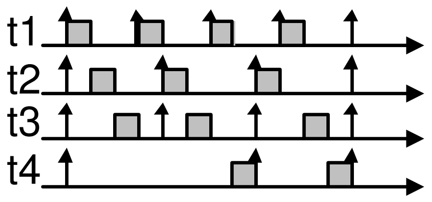


Figure 10: Periodic and aperiodic memory access scheduled with fix priority assignment.

Some tasks may require that the memory access is done in the clock cycle direct after the request. An example is a read operation that happens every three clock-cycles but the result is used in the next clock-cycle. This can be modeled by setting the deadline $D = 1$. Multiple tasks with accesses of $D = 1$ are unfeasible, when the requests occur at the same moment (fig.11, left). Often it may be sufficient if a task operates with a certain rate but is started with a small delay (e.g. a audio-decoder). In this case a phase (delayed start time) can be added to some tasks to make the schedule of the memory accesses feasible (fig.11, right). This approach is limited since it is necessary that all periods are common factors of the largest period.

The examples above have shown how multiple tasks can use the same memory bank without disturbing each other. Up to now, the methods are based on a fix priority schedule which is supported by the first prototype of our MMU. If the memory accesses of a task set are not feasible, some extended techniques can be used:

- The required memory of some task can be located to a separate RAM bank.
- The earliest deadline first method using dynamic priority assignment can be applied.
- The tasks can be modified to extend their deadlines. For example a task that already knows the address of the next read access can pre-load the data to a *temp* register. This can extend the deadline from $D = 1$ to $D = P$.

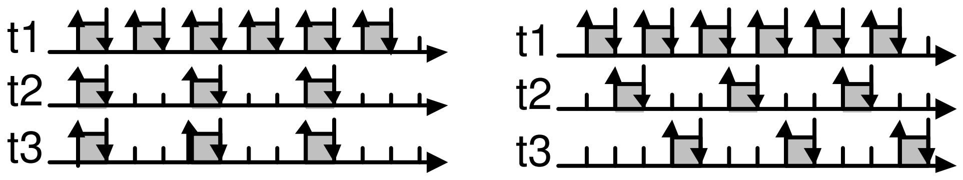


Figure 11: left: Unfeasible schedule of tasks with periodic access. right: Same task set becomes feasible due to adding appropriate phase to some tasks.

4 Conclusion and Future Work

This paper presented a concept of an operating system for computer platforms, which use FPGAs as central computational resource. Such platforms can be useful for a wide range of applications where FPGAs show better characteristics than microprocessors. Several concepts have been adopted from OS of conventional computer platforms, making the system more convenient to use: Applications can be developed and compiled (synthesized) independent of each other and with little dependency on the platform. The use of devices drivers and OS services allows the application to be implemented on a high level of abstraction. Due to the off-line and on-line management of the system resources, statical and dynamical sets of tasks can be executed. Since there is no microprocessor, the entire OS is executed on the FPGAs.

A special concern has been taken to the management of memory. A MMU has been introduced and implemented, which allows multiple tasks to access the same memory bank. Virtual addressing as well as dynamic memory allocation is supported. To resolve access conflicts, the MMU can schedule the memory requests based on fix priorities. This allows periodically accessing tasks to meet their timing constraints.

Future work will deal with the refinement, implementation and overhead analysis of the OS concepts. This includes the task execution model and the placement of tasks. The techniques of the memory management will be extended to satisfactorily deal with multiple memory banks and to improve the scheduling using dynamic priority techniques.

References

[BD01] Brebner, G. und Diessel, O.: Chip-based Reconfigurable Task Management. In: *Field-Programmable Logic and Applications (FPL'01)*. Springer-Verlag, Berlin. 2001.

[BKS00] Bazargan, K., Kastner, R., und Sarrafzadeh, M.: Fast template placement for reconfigurable computing systems. *IEEE Design and Test of Computers*. March 2000.

[Br96] Brebner, G.: A virtual hardware operating system for the xilinx xc6200. In: *Int. Workshop on Field-Programmable Logic and Applications (FPL)*. 1996.

[De96] DeHon, A.: *Reconfigurable Architectures for General-Purpose Computing*. PhD thesis. Massachusetts Institute of Technology. 1996.

[JTY$^+$99] Jean, J. S. N., Tomko, K., Yavagal, V., Shah, J., und Cook, R.: Dynamic reconfiguration to support concurrent applications. *IEEE Trans. on Computers*. June 1999.

[LP02] Lim, D. und Peattie, M.: *Xilinx Application Note XAPP290: Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations*. May 2002.

[TFS00] Teich, J., Fekete, S., und Schepers, J.: Optimization of dynamic hardware reconfigurations. *The J. of Supercomputing*. 19(1):57–75. May 2000.

[WK01] Wigley, G. und Kearney, D.: The Development of an Operating System for Reconfigurable Computing. In: *Proceedings IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*. IEEE CS Press, Los Alamitos, CA, USA. 2001.

[WP03] Walder, H. und Platzner, M.: Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations. In: *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*. S. 284–287. CSREA Press. June 2003.