Algorithmic Skeletons for the Programming of Reconfigurable Systems

Florian Dittmann

Heinz Nixdorf Institute, University of Paderborn
Fuerstenallee 11, 33102 Paderborn, Germany

Abstract. Reconfigurable hardware such as FPGAs combines performance and flexibility, two inherent requirements of many modern electronic devices. Moreover, using reconfigurable devices, time to market can be reduced while simultaneously cutting the costs. However, the design of systems that beneficially explore the reconfiguration capabilities of modern FPGAs is cumbersome and little automated. In this work, a new approach is described that starts from a very high level of abstraction, so-called *algorithmic skeletons*, and exploits the additional information of this level of abstraction to beneficially execute on reconfigurable devices. Particularly, the approach focuses on dynamic run-time reconfiguration on partially reconfigurable FPGAs. As a first introduction to this approach, we consider stream parallelism paradigms including their composition.

1 Introduction

Flexibility and performance are demanding requirements of modern computing systems. Reconfigurable devices offer these requirements as they compute in parallel while still being adaptable (e. g. [1,2]). However, these benefits are cumbersome to explore, particularly, if dynamic reconfiguration shall be exploited. Despite an increasing number of modern FPGAs providing partial run-time reconfiguration—two core requirements for dynamic reconfiguration—methods that allow to exploit these additional flexibilities are rarely found. Nevertheless, some work has been done that proofs the benefit of fine grain granularity and high adaptability of FPGAs, e. g. in [3,4,5,6,7,8].

To eventually exploit the potentials, the cumbersome details of partial run-time reconfiguration should be transparent. We therefore need to offer dynamic reconfiguration on a high level of abstraction to easily gain the benefits of partially reconfigurable systems. These benefits are most likely if the design is done in an FPGA aware manner, i. e., close to the technical (hardware) characteristics of the FPGAs. As the latter is challenging for the application oriented designer, we propose to raise the level of abstraction by using so-called *algorithmic skeletons*. Algorithmic skeletons are programming templates that guide designers to efficiently implement algorithms by separating the structure from the computation itself. In reconfigurable systems, partial run-time reconfigurability thus becomes transparent for the algorithms.

As an introduction to the concept, we show how stream parallelism of applications executed on FPGAs can be abstracted using algorithmic skeletons. On basis of the abstraction, a run-time reconfiguration manager can successfully combine the execution of several—also different—skeletons on one FPGA during the same time.

R. Obermaisser et al. (Eds.): SEUS 2007, LNCS 4761, pp. 358–367, 2007.
© IFIP International Federation for Information Processing 2007

This work is organized as follows: In the next section, we review related work. Section 3 formulates the problem, while Sect. 4 conceptually describes the proposed solution. In Sect. 5, we refine the concept proposed by detailing three skeletons of the stream parallel computing paradigm. Dynamic reconfiguration by virtue of algorithmic skeletons is discussed in Sect. 6. Finally, we conclude and give an outlook.

2 Related Work

In the literature, we find some works on designing reconfigurable systems on a higher level of abstraction than hardware description languages (HDLs). Most of these works do not target partial run-time reconfigurable systems. Additionally, the models proposed assume the designer to have reasonable knowledge of the system under development.

The work of DeHon *et al.* [9] on design patterns for reconfigurable systems is a sophisticated approach on providing canonical solutions to common and recurring design challenges of reconfigurable systems and applications. The authors intend on providing a mean to crystallize out common challenges of reconfigurable system design and the typical solutions. However, their work focusses more on providing a layer of abstraction to the reconfigurable systems community than to application engineers.

Some years earlier, SCORE (Stream Computations Organized for Reconfigurable Execution) was proposed in [10]. The approach focusses on providing a unifying compute model to abstract away the fixed resource limits of devices. Therefore, the resources are virtualized, which can ease the development and deployment of reconfigurable applications. Again, the addressees of the SCORE approach are mainly reconfigurable computing engineers.

Modern languages for embedded systems like SystemVerilog or SystemC also aim at raising the level of abstraction. These approaches can be seen as extended HDLs that introduce design principles to the hardware world, such as re-use, polymorphism, etc. For example, SystemC as language to model dynamic reconfigurable hardware is used in [11]. However, the languages are often used for simulation only and the generation of executable code is challenging.

Further approaches propose an operating systems for reconfigurable systems or FPGAs, respectively, e. g. [7,12,13,14]. These approaches focus on providing the reconfigurable fabric to tasks via the abstraction layer of the operating system. The benefit of these approaches can be a predictable behavior of the executed task. Operating systems, however, seldom consider structure and behavior of the algorithms to be computed.

Finally, in low-level hardware design, [15] focus on a high-level hardware description called hardware skeletons. Considering the idea of separation of structure from the algorithm, this approach is closest to our work. Moreover, the authors motivate their work similar to us, i. e., an increase of abstraction in order to open the field of hardware design to a broader audience. However, the amount of skeletons is very limited and they are still very low-level and will often be too far away from algorithm designers. Moreover, we do not find the paradigm of reconfigurability in their work.

To summarize, all these abstracting approaches barely consider partial run-time reconfiguration and therefore lack the possibility to make the cumbersome details of reconfigurable systems transparent to the application designer.

3 Problem Definition

The design of applications for the execution on partially run-time reconfigurable systems is twofold. On one hand, FPGAs fundamentally are hardware that can be programmed and whose configuration may change over time. Therefore, we need a firm background in hardware design, including communication and I/O requirements. We also have to respect the critical path information, clock skew, etc. Moreover, partially reconfigurable FPGAs require to consider the modification of hardware over time.

On the other hand, the application design is driven by achieving high performance and short time to market. Designers therefore explore the theory behind applications and search for algorithms that server the problems best. Moreover, they try to abstract from the execution platform, mostly due to reasons of programmability and portability. Partial run-time reconfiguration becomes a feature that should be beneficially for the performance of the algorithm. The details of hardware and FPGAs thereby are of secondary focus, as development takes place more in the terms of the software world, even if special requirements of embedded systems are respected.

Synthesis from behavioral problem description to reconfigurable hardware targets this issue. However, in the domain of partial run-time reconfigurable hardware, automatic synthesis still lacks good results. Furthermore, if iterative design due to performance evaluation is required, or portability is an issue, we require a more suitable design methodology that supports designers on a high level of abstraction.

4 Problem Solution

We propose to use algorithmic skeletons as bridge between circuit design and application development for FPGAs. Algorithmic skeletons therefore are offered as a library that is used by the algorithms of the application under development. The usage of algorithmic skeletons constrains the design of algorithms to a set of templates. However, we can extract valuable information for dynamic reconfiguration from these templates.

4.1 Algorithmic Skeletons

Algorithmic skeletons were introduced by Cole [16]. They separate the structure of a computation from the computation itself. Originally, the application domain of algorithmic skeletons are parallel machines or cluster computers. In particular, the skeletons free the programmer from the implementation details of the structure, such as how to map it to the available processors. By providing a structured management of parallel computation, they can be used to write architecture independent programs, shielding application developers from the details of a parallel implementation.

Algorithmic skeletons are similar to higher order functions of functional languages for conventional imperative languages. Concerning design space exploration, skeletons and their level of abstraction enable to explore a variety of parallel structurings for a given application. A clean separation between structural aspects and the application specific details is realized by virtue of algorithmic skeletons. Thanks to the structural information provided, static and dynamic optimization of implementations is possible.

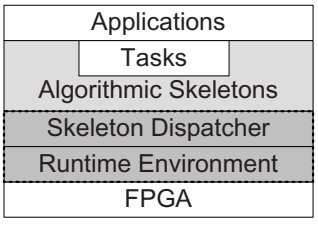

Fig. 1. Layer model

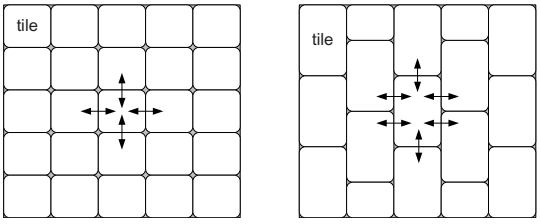

Fig. 2. Two run-time execution environments

The purpose of every skeleton is to abstract a pattern of activities and their interactions. They provide a means of implementation, which separates them from design patterns. The latter are mostly used during the design phase and offer only orientation for the final implementation. Due to their proximity to a run-time environment, algorithmic skeletons allow us to exploit the performance offered by the processing system.

Consequently, there has to be a balance between generality (allowing re-use for different architectures and user kernels) and specificity (for efficient implementation and interfaces to the user kernels). There also is the so-called trap of universality, i. e., providing a skeleton that is generic in itself and can be used if no other skeleton might fit. Such a skeleton would increase the complexity of a run-time environment. In order to avoid this trap, there is usually the restriction of the acceptable input for a system to a set of valid algorithmic skeletons only, see also [17,18]. In case of modern FPGAs, we can also overcome this gap by exploiting soft or hard core CPUs. These general purpose processors can execute any algorithm due to their Turing completeness.

4.2 Application in Reconfigurable Systems

Reconfigurable computing on FPGAs basically is similar to processing on parallel systems, as execution of algorithms on hardware like FPGAs also means processing in parallel. When reconfiguring FPGAs, we usually define exchangeable regions and apply different modules to these regions. Several such regions can be marked on the same FPGA. These regions are comparable to the nodes of a computing cluster. The inter-module communication, so still a challenging research area, enables various ways of data exchange. Thus, we see broad similarities to parallel computing in the sense of algorithmic skeletons. We can distribute applications into the regions as it is done in the parallel computing domain. For efficient execution and beneficial exploitation of the capabilities, both systems need structure, which is provided by algorithmic skeletons.

Therefore, we use algorithmic skeletons as means of abstraction for partial run-time reconfiguration of FPGAs. The skeletons provide a seminal method to abstract reconfigurable fabrics on a high level. We combine the skeletons into a library. As a first introduction to this new concept, we detail stream parallelism in the next section.

We abstract the general concept by virtue of a layer model, see Fig. 1. Applications, which built the top layer, are described by a set of tasks. These tasks must be implemented using algorithmic skeletons. An execution environment that executes the tasks

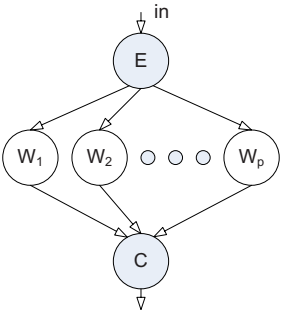

Fig. 3. Farm parallelism

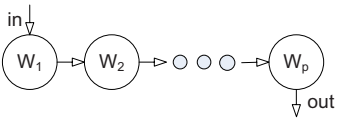

Fig. 4. The pipeline paradigm

on an FPGA accepts the tasks described by a set of skeletons only. The set of skeletons is processed by a dispatcher that is deeply connected to its execution environment.

4.3 Execution Environment

Concerning the practical realization of a suitable run-time environment for the execution of the set of skeletons on an FPGA, we consider a tiled system. As we focus on homogenous FPGAs in this work, each tile comprises similar logic resources. However, we still consider two different tile arrangements: the first being a purely quadratic organization, while the second one offers more direct communication possibilities due to an underlying hexagonal structure (see Fig. 2).

Xilinx Virtex 4 devices support the proposed execution environments as they support 2D style partial reconfiguration. Furthermore, on these devices, the external communication, i. e. the I/O pads, is separated and not part of a slice. With the advances in FPGA design, more sophisticated run-time environments are possible.

5 Stream Parallelism

Stream parallelism may be the closest idea of parallel computing that matches the ideas of execution on FPGAs [19]. Stream computation can be described as applying $f : \alpha \rightarrow \beta$ on a stream of input values $a_1, a_2, \ldots$. The idea is to exploit the parallelism within the computation of f on different (and unrelated) elements of the input stream. As an example, we can consider a vision system that explores images. The images enter the system abstracted as a stream and must be handled differently.

5.1 Farm Paradigm

An algorithm that computes the same f on all of the elements of a stream $a_1, a_2, \ldots$ exploits the farm paradigm. The computations $f(a_1), f(a_2), \ldots$ can be executed in parallel using a pool of parallel processing modules. Figure 3 depicts the concept. The major characteristic to observe is that the functions can be executed independently of each other as they are all operating on a different data set.

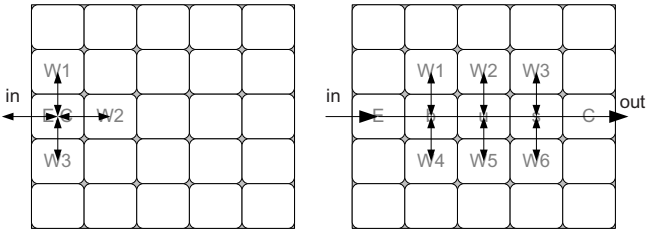

Fig. 5. Two possible execution schemes of applications using the farm skeleton

As an application example, we can assume a stream of two video channels that should be output alternatingly to one single channel. However, the switch between the two channels should not be abrupt but smoothly, i. e., a fading between the two channels. We thus have a function f that is applied on a stream of three input values $\langle x_1, y_1, c_1\rangle, \langle x_2, y_2, c_2\rangle, \ldots$, while x_1 and y_1 denote the two video streams and c_1 the dominance of the one stream over the other (an increasing/decreasing number of e. g., 8 bit width). Both streams arrive at the node E which distributes the single images to the worker processes $W_1, W_2, \ldots, W_n$ adding the number c_i. These functions can be computed independently of each other in different worker nodes W_i. The results then are propagated to the combining node C that forwards the stream to the video screen.

If we describe our algorithm using a skeleton for the farm paradigm, the structure of the application is given. Thus, we know how to execute the algorithm on the execution device. First, we can derive a meaningful placement of the algorithm that serves both the requirements of the farm and the characteristics of the FPGA. Figure 5 shows how the skeleton can be mapped in two different ways. In the left approach, we use the same tile for the input and output of the nodes. However, if we rely on direct communication links only, the number of possible worker tiles is limited. Therefore, the right approach of Fig. 5 spans the farm skeleton over the whole width of the FPGA.

Dynamic run-time reconfiguration is needed if the amount of worker modules should be adapted during run-time. External stimuli therefore could be a requirement to adapt the quality of service, etc. Further details are discussed in Sect. 6.

To summarize the farm skeleton, a structural concept is given that allows to distribute workers of an application on different tiles of a partially and run-time reconfigurable FPGA. The execution of the workers including their reconfiguration is part of the run-time environment and its dispatcher. However, by describing an application on basis of the farm skeleton, the number of workers is not set. Depending on the resources available, a different quality of service can be realized. The optimal solution, i. e., a solution that avoids the blocking of workers, etc. due to overload conditions, must be derived carefully by evaluating the execution times of the function f and the distribution time of the initial node E.

5.2 Pipeline Paradigm

The pipeline paradigm comprises a composition on n functions $f_1 \ldots f_n$ such that

$$f_1 : \alpha \rightarrow \gamma_1, \ldots, f_i : \gamma_{i-1} \rightarrow \gamma_i, \ldots, f_n : \gamma_{n-1} \rightarrow \beta \tag{1}$$

Figure 4 shows the concept as a graph.

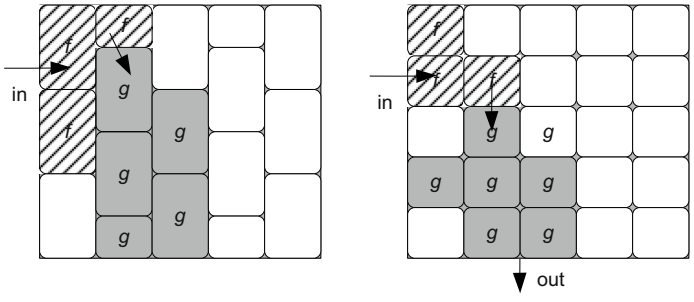

Fig. 6. Two realizations of a pipeline with different area requirements

As an example within our image processing environment, we consider a scenario of a stereo vision system. We receive the input of two cameras $\langle x_1, y_1\rangle, \langle x_2, y_2\rangle, \ldots$ and want to extract valuable information out of the system. We therefore compute the composition of two functions $f \circ g$. Function g will result in a combination of the two images, having the pixel combined into the means ($g(\langle x_i, y_i\rangle) = z_i$), while f will produce the histogram on the resulting image z_i. The functions f and g can be executed in parallel each on a subsequent data set, thus exploiting pipeline parallelism.

In Fig. 6, we depict two possible realizations assuming the second function f to consume more area than function g. Here we can see that different stages can consume more area than available on one single tile by simply combining tiles. The dispatcher of the run-time environment may react on the different requirements of the functions within the pipeline. If enough area is available, the dispatcher may also built up a second pipeline in parallel in order to increase the throughput of the systems.

Describing a problem using the pipeline skeleton, we can further exploit the characteristics of stream processing. As the stages of the pipeline get activated in sequence, we can decrease the reconfiguration latency of the overall system. We successively load the bitstreams of the pipeline stages in their order given. After reconfiguring the first stage, this stage may start its execution before the complete pipeline is loaded. The same holds for the subsequent stages. Thus, the fastest possible response time can be guaranteed. Additionally, if less area than required by the stages is available, we may apply hardware virtualization. Therefore, only parts of the overall pipeline are loaded on the FPGA at the same time. These parts may also perform block processing of a block of input sets in order to hide the reconfiguration overhead, which is still in the range of milliseconds on modern FPGAs.

A further approache to improve the behavior of a pipeline streaming algorithm is to identify the bottleneck stage. In order to reduce the impact of this stage, we can provide a functionality to map this stage on a tile comprising of specific computation resources (assuming a heterogenous FPGA). Alternatively, we might provide critical stages in different implementation variants that can be tested in a design space exploration that explores different implementations of the pipeline skeleton. We then select the combination of these stages that offer the best overall performance.

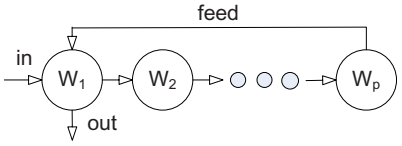

Fig. 7. Stream iterative paradigm

5.3 Stream-Iterative Paradigm

In the stream-iterative paradigm, we have a number of functionally equivalent stages. This number of stages may depend on the input values and is generally unknown before execution. We can view the stream-iterative paradigm as a tail recursive function f that comprises of a finite result x if a boolean function $c(x)$ computes true, or a recursive call $f(g(x))$ otherwise. We can compute such a problem in parallel by a pipeline including a stage for each recursive call of f. Figure 7 depicts the concept. In order to implement such an unbounded pipeline on an FPGA with limited resources available, we emulate the unbounded pipeline by folding it on a chain of processes of a fixed length.

As an example, we can consider again a stream of images that are processed by a filter in each of the stages. The processing will go on until no further refinement of the image is possible and the final result is sent to the output.

6 Dynamic Reconfiguration

The above presented paradigms can be composed to built more complex parallel structures. In Fig. 8, we show how different skeletons can be executed on the same FPGA, exploiting a multi task environment. Depending on the specific needs (quality of service, etc.) of the applications behind the skeletons, we can react and dynamically adapt the purpose and organization of the tiles of our execution environment.

Such a dynamic reconfiguration means the adaptation of a device during run-time. In particular, the amount and shape of tasks that shall be executed on a run-time environment are not known at design time. When realizing such a behavior without any abstracting layers on top of an FPGA, we would have to cope with fragmentation and on-line routing issues that can be tremendously challenging.

The implementation of applications by virtue of algorithmic skeletons enables a sophisticated and dynamic execution of tasks on FPGAs. The run-time environment allows us to load tasks which are available on the basis of algorithmic skeletons onto the FPGA. As the usage of algorithmic skeletons enforces the applications to be well-formed, we thereby can prevent fragmentation of the devices and guarantee communication requirements. Additionally, the quality of service may be considered.

In the example depicted in Fig. 8, we first assume a scenario where a farm skeleton is executed in the left side of the FPGA and a stream-iterative skeleton occupies the right side of the FPGA. The former one can use four worker tiles, while the latter has eight worker tiles on its dispose. At some point in time, a new application requests to enter the system. We can decrease the pipeline of the stream-iterative skeleton, thus freeing

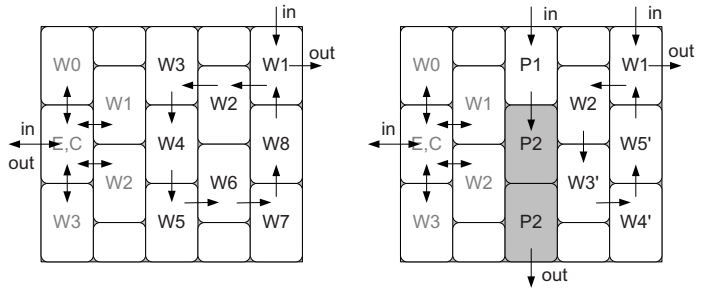

Fig. 8. Combination of different skeletons

area in the middle of the FPGA. This area is then used to execute a new application that ist implemented referring to the pipeline skeleton.

In the example, the execution environment is fixed, as it provides the computational resources and the communication for its set of skeletons. The dispatcher accepts a set of skeletons only. On basis of the information of the skeletons, we can take care of connections, etc. The combination of dispatcher and specific run-time environment allows us the execution of a set of skeletons. We can execute any algorithm on this run-time environment irrespectively of its behavior and size, as long as the algorithm can be implemented by virtue of some of this environment's skeletons. If the area requirements exceed the size of the FPGA, we can apply hardware virtualization as described above. As a drawback, we only serve applications which are implemented as skeletons that the execution environment supports.

The design of applications by virtue of algorithmic skeletons allows us to react on changing needs of the whole system and of a single application of the system. In particular, if the quality of service must be increased, we can demand additional resources.

7 Conclusion

In this work, we have introduced algorithmic skeletons for dynamic reconfigurable computing. Algorithmic skeletons separate structure from the behavior of an algorithm. By providing a library of skeletons to implement applications for reconfigurable systems, we can beneficially explore partial run-time reconfiguration of reconfigurable fabrics. Therefore, solutions, i. e., hardware realizations, of the skeletons are applied to various applications. We have introduced the field of stream parallelism comprising of the farm, pipeline and stream-iterative paradigm. In general, the approach is a hopeful mean to provide an interface between the hardware platform (FPGA) and applications. Moreover, additional benefits are possible if a composition of skeletons is used.

We currently broaden the library of algorithmic skeletons to offer also data prallelism. Furthermore, we want to consider heterogeneous FPGAs, as the additional resources of such fabrics facilitate improved solutions for specific applications that we hope to also cover by algorithmic skeletons. As a final outlook, also coarse-grain reconfigurable devices as execution environments may be taken into account.

References

1. DeHon, A., Wawrzynek, J.: The case for reconfigurable processors (1997)
2. Compton, K., Hauck, S.: Reconfigurable Computing: A Survey of Systems and Software. ACM Computing Surveys 34(2), 171–210 (2002)
3. Ganesan, S., Vemuri, R.: An Integrated Temporal Partitioning and Partial Reconfiguration Technique for Design Latency Improvement. In: DATE '00. Proceedings of the IEEE Design, Automation and Test in Europe, Paris, France (2000)
4. Diessel, O., ElGindy, H., Middendorf, M., Schmeck, H., Schmidt, B.: Dynamic Scheduling of Tasks on Partially Reconfigurable FPGAs. IEE Proceedings – Computer and Digital Techniques (Special Issue on Reconfigurable Systems) 147(3), 181–188 (2000)
5. Horta, E.L., Lockwood, J.W., Taylor, D.E., Parlour, D.: Dynamic hardware plugins in an FPGA with partial run-time reconfiguration. In: DAC '02. Proceedings of the 39th conference on Design automation, pp. 343–348. ACM Press, New York (2002)
6. Li, Z., Hauck, S.: Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In: FPGA '02. Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays, pp. 187–195. ACM Press, New York (2002)
7. Steiger, C., Walder, H., Platzner, M.: Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. IEEE Trans. Comput. 53(11), 1393–1407 (2004)
8. Danne, K., Bobda, C., Kalte, H.: Run-time Exchange of Mechatronic Controllers Using Partial Hardware Reconfiguration. In: Cheung, P.Y.K., Constantinides, G.A. (eds.) FPL 2003. LNCS, vol. 2778, Springer, Heidelberg (2003)
9. DeHon, A., Adams, J., DeLorimier, M., Kapre, N., Matsuda, Y., Naeimi, H., Vanier, M.C., Wrighton, M.G.: Design patterns for reconfigurable computing. In: FCCM, pp. 13–23 (2004)
10. Caspi, E., Chu, M., Huang, R., Yeh, J., Wawrzynek, J., DeHon, A.: Stream computations organized for reconfigurable execution (score). In: Grünbacher, H., Hartenstein, R.W. (eds.) FPL 2000. LNCS, vol. 1896, pp. 605–614. Springer, Heidelberg (2000)
11. Antti Pelkonen, K.M., Cupák, M.: System-Level Modeling of Dynamically Reconfigurable Hardware with SystemC. In: Proceedings of International Symposium on Parallel and Distributed Processing (Reconfigurable Architectures Workshop), pp. 174–181 (April 2003)
12. Brebner, G.J.: A Virtual Hardware Operating System for the Xilinx XC6200. In: Glesner, M., Hartenstein, R.W. (eds.) FPL 1996. LNCS, vol. 1142, pp. 327–336. Springer, Heidelberg (1996)
13. Danne, K.: Operating Systems for FPGA Based Computers and Their Memory Management. In: ARCS 2004.Organic and Pervasive Computing, Workshop Proceedings. GI-Edition Lecture Notes in Informatics (LNI), vol. P-41, Köllen Verlag (2004)
14. Walder, H., Platzner, M.: A Runtime Environment for Reconfigurable Hardware Operating Systems. In: Becker, J., Platzner, M., Vernalde, S. (eds.) FPL 2004. LNCS, vol. 3203, pp. 831–835. Springer, Heidelberg (2004)
15. Benkrid, K., Crookes, D.: From application descriptions to hardware in seconds: a logic-based approach to bridging the gap. IEEE Trans. VLSI Syst. 12(4), 420–436 (2004)
16. Cole, M.I.: Algorithmic Skeletons: Structured Management of Parallel Computation. Pitman/The MIT Press, London, UK/Cambridge, Massachusetts, USA (1989)
17. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Comput. 30(3), 389–406 (2004)
18. Rabhi, F.A., Gorlatch, S.: Patterns and Skeletons for Parallel and Distributed Computing. Springer, Heidelberg (2002)
19. Pelagatti, S.: Structured development of parallel programs. Taylor & Francis, Inc., Bristol, PA, USA (1998)