

# EVALUATION OF RAY CASTING ON PROCESSOR-LIKE RECONFIGURABLE ARCHITECTURES

T. Oppold, T. Schweizer, T. Kuhn, W. Rosenstiel \*

U. Kanus, W. Straßer †

WSI/TI  
Universität Tübingen  
72076 Tübingen, Germany

WSI/GRIS  
Universität Tübingen  
72076 Tübingen, Germany

## ABSTRACT

Ray casting, a real-life application for the visualization of 3D scientific and medical data, demands both high performance and high flexibility, and is likely to be used in future mobile devices, where low power is an important issue. In this paper, we show that implementing ray casting on coarse-grained, highly reconfigurable architectures, satisfies these conflicting requirements much better than traditional architectures.

## 1. INTRODUCTION

Visual computing, i.e., the visual representation of application data, is becoming a more and more important tool for understanding the huge amount of data provided by current and future medical and scientific applications. Direct volume rendering (DVR) is a visual computing method that is used to display 3D volumetric data, for example data from medical scanners (CT, MRI) or physical simulations, on a 2D screen.

Several algorithms for volume rendering have been proposed, of which ray casting [1] is the most popular. Ray casting is very demanding in terms of computational power and memory bandwidth and therefore requires the use of hardware acceleration for real-time performance. An ASIC solution can deliver the required performance [2], but is not flexible enough to keep up with the ongoing development of extensions to the basic ray casting algorithm, for example the inclusion of pre-integrated rendering [3], and non-photo-realistic rendering techniques [4].

Software implementations, like the UltraVis system [5], have to make heavy use of the multimedia extensions (e.g. SSE2 or 3DNow) of current CPUs and a cache optimized memory layout to achieve interactive performance, which is tedious to code and prone to fail for even slight changes in processor/memory architecture.

Using FPGAs, like the VIZARD-II architecture [6], allows for a more flexible implementation compared to an ASIC. However, the implementation process is nearly as complex and thus not suitable for the rapid inclusion of new features. Additionally, FPGAs are not suitable for application areas where low power consumption is a requirement.

The FPGAs commonly used today can be classified as fine-grained reconfigurable devices, most suitable for random logic. For a known application domain, coarse-grained architectures with hardwired functional units can lead to improvements in terms of performance and power dissipation. Such hardwired components can be found in state-of-the-art FPGAs, which makes them more efficient for DSP applications. Another benefit is the reduced amount of configuration data required, such that it is easier to store multiple configurations on-chip. This also leads to shorter reconfiguration cycles.

In this paper, we focus on architectures, for which frequent reconfiguration is part of the regular execution. We call these architectures *processor-like* reconfigurable architectures. Commercial examples for such architectures are NEC DRP [7], PACT XPP [8] and the FMCA and GPMCA accelerators of the Intel Reconfigurable Communications Architecture (RCA) [9].

For the evaluation of processor-like reconfigurable architectures we have defined the CRC model (Configurable Reconfigurable Core) [10]. This abstract model is refined in an iterative process in order to create instances of the CRC model, ready for implementation in silicon. The iterative refinement process is driven by the application requirements, as well as the support for automated mapping of algorithmic descriptions of the application.

In the following sections, we first give a short description of the ray casting algorithm. We then describe the application of the iterative process to find two architectures with different optimization goals. In Section 4, we discuss the results for the proposed architectures compared to an FPGA implementation.

\*[oppold, tschweiz, kuhn, rosenstiel]@informatik.uni-tuebingen.de

†[kanus, strasser]@gris.uni-tuebingen.de

## 2. RAY CASTING

The ray casting algorithm is usually implemented in a pipeline, consisting of five major stages:

**Ray Setup:** For each pixel of the view plane, the viewing ray from the eye position through the view plane pixel towards the volume data is calculated and tested for intersection with the volume. If an intersection was found, the ray entry point and the ray increment is passed on to the ray traversal stage.

**Ray Traversal and Voxel Fetch:** Each ray that hits the volume data is traversed in discrete steps, given by the ray increment, until it leaves the volume. At each new sample location along the ray, a  $2 \times 2 \times 2$  neighborhood of voxels required for resampling the volume data at this location has to be fetched from memory. Determining the memory addresses of these voxels depends on the storage layout of the volume data.

**Resampling:** A filter kernel is applied on the volume data to obtain the sample value at the current sample location. Commonly used filter kernels are trilinear interpolation and nearest neighbor interpolation.

**Classification and Shading:** Classification is used to assign material properties to a sample value. This is done by using the interpolated sample value as an index into a look-up table. Shading is often performed using the Phong model [11]. Alternatively, non-photo-realistic shading techniques [4] are used to enhance the visual perception of volume data.

**Compositing:** In the compositing stage, classified and shaded samples are accumulated along a ray, yielding the red, green, blue, and alpha channel of the final pixel on the image-plane.

## 3. ARCHITECTURE EXPLORATION

For the evaluation of ray casting on processor-like reconfigurable architectures we have chosen the pipeline stages *voxel fetch*, which is memory bandwidth limited, and *resampling*, the most computational intensive stage [6]. These two pipeline stages represent the requirements of ray casting well enough to make a comparison to FPGAs without evaluating the entire application on both architectures, which is too time consuming for a first assessment.

An important aspect of our design flow for finding a good architecture for ray casting is that the application development can be done in relatively short time using well known programming languages. Therefore we used an implementation of ray casting formulated in the C language, which uses fixed point arithmetic but other than that it is a

typical software implementation that can be compiled with a generic C compiler for testing. No assumptions were made about the target architecture in order to make use of specific architectural features.

To map applications onto processor-like reconfigurable architectures we develop a tool chain that translates algorithmic C descriptions into the binary format that can be executed by instances of our CRC model. This chain is not completely implemented yet, but the low control complexity of the chosen pipeline stages allows to perform the missing steps manually with reasonable effort. The low control complexity also facilitates manual translation to the register-transfer level which we used for the mapping onto an FPGA.

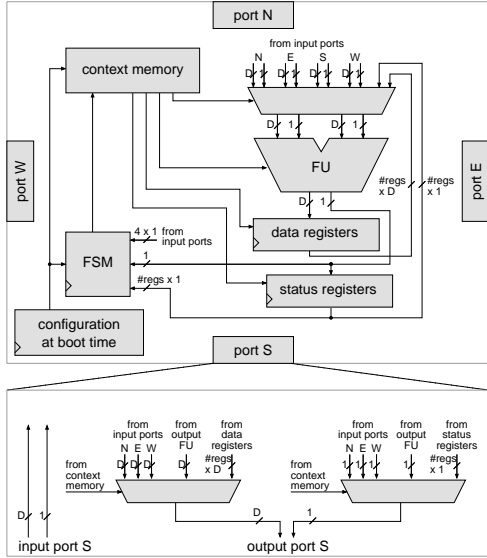
### 3.1. Architecture for High Throughput

As an initial architecture for the evaluation we used an array of PEs that are based on the PEs presented in [12]. We are aware that this PE design is obviously not optimal for a final implementation, but it provides easy scalability required for an early assessment. As depicted in Fig. 1, each of these PEs provides a functional unit (FU) that supports all operators of the C language except for  $/$  and  $\%$ , registers to store data and 1-bit status signals, a context memory, and a configurable finite state machine (FSM).

The connectivity between the PEs is realized by a nearest neighbor interconnect network that allows each PE to route data and status signals from and to its direct neighbors and to simultaneously execute an operation in the FU. The input and output ports of the PEs at the border of the array are used for I/O purposes and to attach memory.

An entry in the context memory determines the context of the current clock cycle, i.e., the operation performed by the FU, source and destination of the operation (either from/to the ports of the PE or registers), and the routing between neighboring PEs. The switching between contexts can occur at every clock cycle, controlled by the FSM whose inputs are connected to the status signals of the PE. The FSM is implemented as a Medvedev machine, i.e. the context is directly determined by the state. We see the FSM as a helpful feature to map algorithmic descriptions onto the architecture since it allows to realize the state transitions generated by the compiler efficiently. The content of the context memory and the FSM state transitions are configured similar to an FPGA.

To refine the initial architecture we analyzed the source code of the ray casting implementation. This analysis reveals that 32-bit wide arithmetic operations and 16-bit x 16-bit multiply-operations producing 32-bit results are needed throughout the algorithm. We therefore set the width of the data path (D in Fig. 1) to 32 bit and adapted the multipliers to the required widths. To maintain enough flexibility for different implementations of the ray casting algorithm we did not analyze the operations of each pipeline stage in-



**Fig. 1.** A PE of the initial architecture. D denotes the width of the data path that can vary for different implementations.

dividually to determine specialized FUs for each stage but considered only homogeneous arrays, i.e., all PEs of the architecture are identical and they are distributed uniformly over the array.

In the next step we determined the number of PEs for the voxel fetch stage and their spatial arrangement. The software style implementation of the voxel fetch stage contains 3 nested loops. When these loops are unrolled and optimizations like common sub-expression elimination are applied, the resulting basic block contains 15 add-operations and 2 multiply-operations. Due to the data dependencies, parallel execution of the 17 operations is limited. To achieve a high throughput rate even so, the operations of the voxel fetch stage were broken into 4 additional (super-)pipeline stages such that each stage contains between 3 and 8 operations that can be executed in parallel. The operations of each of the first three stages can be mapped to a column of 4 PEs or less. The last stage can be distributed over 2 columns of 4 PEs. Therefore, an array of 5 x 4 PEs is a reasonable arrangement for super-pipelined mapping of the voxel fetch stage.

With the PEs used so far the operations can be mapped to the FUs of the PE array. But the interconnect network provides only one bidirectional 32-bit data channel between two neighboring PEs. With these resources it is not possible to establish the needed connectivity between the PEs of the 5 x 4 array. To overcome this bottleneck a second set of data channels for the nearest neighbor network was introduced in a new architecture. With these additional resources the mapping of the voxel fetch stage was possible for the 5 x 4 array. Additionally, data that is not consumed by the

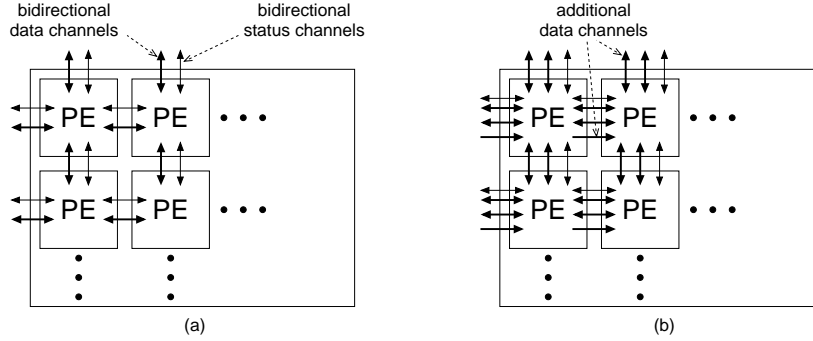
voxel fetch stage must be transported to the resampling stage with the same latency, therefore a data channel with low resource requirements connecting only PEs from left to right was added. Figure 2 shows an overview of the interconnect features of the resulting architecture compared to the initial architecture. The second set of data channels for the nearest neighbor interconnect network is also needed to provide enough bandwidth for the memory that is addressed by the voxel fetch stage. The voxel fetch stage generates 8 memory addresses. To process one sample at each clock cycle, 8 memory blocks with identical sets of volume data are necessary. With the additional resources of the refined architecture the two columns of the last super-pipeline stage provide a total of 8 data ports to transport the addresses to the memory blocks.

The 8 memory blocks are caching a sub-cube of the complete data set, since the complete data set can be quite large.  $16^3$  voxels is a reasonable size. When data is not immediately available in such a memory block, a cache controller, presented in [13], has to fetch voxels from external memory, resulting in a pipeline stall. To connect the memories to the array of PEs, we propose a bus that allows the PEs at the top and bottom of the array to access the memories above or beneath them. Write access on the bus can be controlled by the cache controller that is attached to each memory block.

If the data is available in the memory blocks, one clock cycle after voxel fetch generates the memory addresses, the volume data from the memories is the input of the resampling stage. The resampling is done by *trilinear interpolation* or *nearest neighbor interpolation*. Based on the results of previous stages, one of the two alternatives is chosen dynamically during operation. Processor-like reconfiguration allows it to keep the configurations for both alternatives in the context memory and to switch between them as they are needed. By this temporal mapping, the number of PEs needed to provide both trilinear and nearest neighbor interpolation is the maximum of the needed PEs of both alternatives.

The implementation of trilinear interpolation contains 28 operations in total, composed of subtract and multiply as well as shift-operations for fix point adjustment. To compute the trilinear interpolation each input operand is subject to 12 operations before the resulting sample value is calculated. Accordingly, 12 super-pipeline stages are needed if a high throughput rate is desired. Since the nearest neighbor interconnect network allows not only data to be routed from left to right, i.e. the logical data flow direction, but also from right to left as well as up and down, the 12 super-pipeline stages of trilinear interpolation can be mapped to an array of 10 x 4 PEs.

The implementation of nearest neighbor interpolation uses 3 compare operations to control if-else branches in the source code. One possibility of processor-like reconfig-



**Fig. 2.** Overview of the interconnect resources of (a) the initial and (b) the refined architecture.

urable architectures to realize branches in the control flow is to switch contexts depending on the branch condition. To implement branches within one context, the PEs of the proposed architecture feature a select function. Using these select functions, the nearest neighbor interpolation can be mapped onto the array of  $10 \times 4$  PEs needed for trilinear interpolation easily using one context.

Figure 3(a) shows an overview of the spatial composition of the architecture that we propose for a high throughput rate of the voxel fetch and resampling stages of ray casting. Figure 3(c) depicts the context requirements for each column of the array for the mapping described above. While the super-pipelined implementation of voxel fetch requires only one context during normal operation, two contexts are needed for the two super-pipelined implementation alternatives of the resampling stage. Since the memory architecture may cause the pipeline to be stalled due to cache misses, 2 additional contexts to handle that situation are reserved for each PE resulting in a total of 4 contexts that are needed for each PE in a homogeneous architecture.

### 3.2. Architecture for Low Area

The switching between two alternative parts of an application that are not needed at the same time shows how processor-like reconfiguration can be utilized to reuse the functional units of the PEs in order to save area. Since the switching between contexts in processor-like reconfigurable architectures requires only a fraction of a clock cycle, the imposed performance penalty is relatively low. If the throughput requirements of a device allow it to reduce the performance significantly, e.g. if only still images need to be displayed, the area of the architecture can be reduced by mapping the super-pipeline stages onto different contexts instead of different columns of the array.

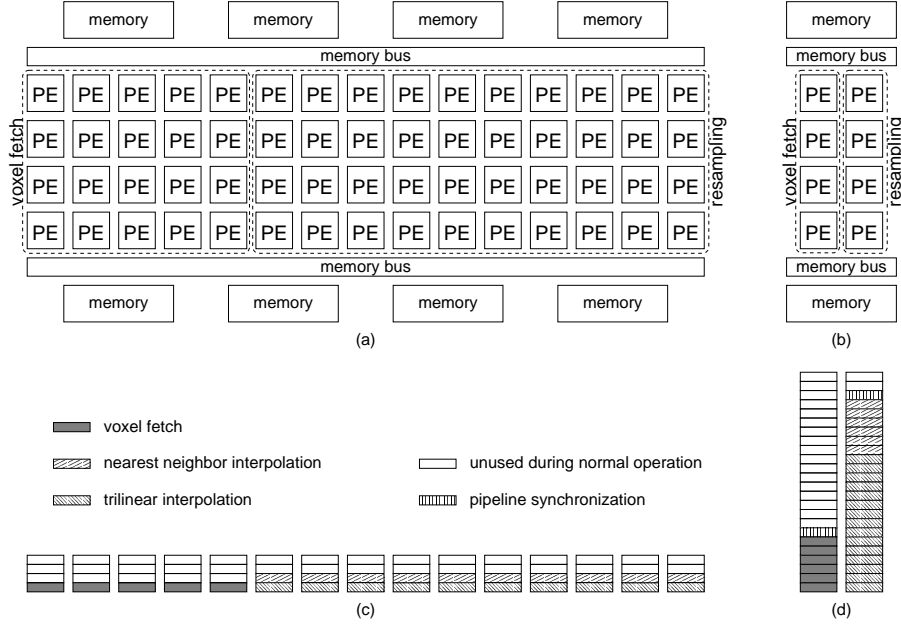
To evaluate an architecture with low area requirements we mapped the voxel fetch and resampling stage also to an array of  $2 \times 4$  PEs as depicted in Fig. 3(b) using one column for each stage. Partitioning a pipeline stage tem-

porally into multiple contexts can be done using the same basic techniques as for partitioning it spatially into super-pipeline stages as described above. Due to the spatial limitation of one column, those super-pipeline stages that were originally mapped to multiple columns must be further partitioned though to fit one column of 4 PEs.

To ensure synchronization between the pipeline stages, both stages must always use the same number of clock cycles to process one sample. For a multi-context mapping, this means that the maximum number of contexts needed by voxel fetch, trilinear interpolation, or nearest neighbor interpolation determines the contexts available for each of them without slowing down the throughput rate. The highest context requirements are imposed by the trilinear interpolation which we mapped to 15 contexts. Accordingly, 15 clock cycles are needed to process on sample and the operations of voxel fetch and nearest neighbor interpolation can be distributed over up to 15 contexts. These 15 contexts can be used to distribute the 8 memory accesses over multiple clock cycles resulting in an architecture that needs only 2 memory blocks instead of 8. Voxel fetch and nearest neighbor interpolation actually require only 6 contexts each. While theoretically there is no need to keep the number of contexts for voxel fetch lower than 15, the number of contexts used by nearest neighbor interpolation must be added to the number of contexts used by trilinear interpolation to determine the total number of contexts. By using one additional context that simply counts 9 down to 0, nearest neighbor interpolation can be done in 7 contexts while keeping the pipeline stages synchronized.

Figure 3(d) depicts the context requirements for the two columns of PEs when a multi-context mapping is performed. As in the case of the super-pipelined implementation, at least 2 additional contexts are reserved for handling pipeline stalls resulting in a total of 24 contexts per PE needed for the multi-context implementation. Besides the different number of contexts the PEs are the same as for the super-pipelined implementation.

Finally, the number of data and status registers must be



**Fig. 3.** Architectures for (a) high throughput and (b) low area. (c) and (d) show the usage of context memory for each column of PEs.

determined for the PEs to fully characterize the two proposed architectures. For both pipeline stages one PE requires at most 5 data registers and 2 status register considering the super-pipelined and the multi-context mapping. Setting the number of registers for data to 7 and for status signals to 3 provides a good balance between headroom for different implementations of ray casting and utilization of the context memory in terms of unused bit combinations in our implementation of the PEs.

#### 4. DISCUSSION OF RESULTS

To estimate the area requirements of the suggested architectures, their power dissipation, and the clock speed at which they can be run, we synthesized the Verilog model of the PEs presented in the previous section and analyzed the resulting gate level design using commercial tools. As the target technology for synthesis we used a 130 nm standard cell library. By defining the target technology and setting the constraints for synthesis, the proposed architectures become distinct instances of the CRC model.

For one PE with 4 contexts, a cell area of  $97,559 \mu m^2$  is estimated. To determine the maximum clock speed, the estimated delays of the PE were used to calculate the longest delay path which includes the time needed for a context switch. For the super-pipelined mapping the longest delay of all super-pipeline stages is 6.12 ns. When run at the maximum clock speed of 163.4 MHz the architecture is capable of processing 163.4 million samples per second. At this

clock speed one PE is estimated to dissipate 6.20 mW based on the default switching activity parameters of the power analysis tool.

With 24 contexts, the estimated area for one PE is  $161,105 \mu m^2$ . For the multi-context mapping, the longest delay over all contexts of both pipeline stages is 6.17 ns. This mapping allows it to process one sample in 15 clock cycles such that 10.8 million samples can be processed when the architecture is run at the maximum clock speed of 162.1 MHz. A power dissipation of 9.34 mW is estimated at this clock speed for one PE. As for the area estimation the higher power dissipation is caused by the increased number of contexts for the multi-context implementation.

Table 1 summarizes the results for the two instances of the CRC model suitable for super-pipelined and multi-context execution of voxel fetch and resampling and compares them to a super-pipelined implementation on a Xilinx Virtex-II FPGA which is fabricated in a comparable technology process. The memory is not considered in the comparison. The FPGA achieves almost the same performance as the super-pipelined implementation on the CRC model with its hardwired functional units. This is due to the fact that the synthesis tool for the FPGA was able to exploit the DSP oriented features like dedicated 18-bit x 18-bit multipliers and fast carry chains, which enable a significant speed-up compared to a pure look-up table based implementation.

The estimated power dissipation of the FPGA is also based on the default parameters for the switching activity. It includes only the figures for the core operated at 1.5 V and

**Table 1.** Summary of the results and comparison to an FPGA implementation.

architecture	clock speed	samples per second	total power dissipation	energy per sample	area
CRC (super-pipelined)	163.4 MHz	163,400,000	372 mW	2277 pJ	60 PEs (5.85 mm <sup>2</sup> )
CRC (multi-context)	162.1 MHz	10,800,000	74.7 mW	6913 pJ	8 PEs (1.29 mm <sup>2</sup> )
Virtex-II 1000 (super-pipelined)	140 MHz	140,000,000	982 mW	7014 pJ	341 slices 9 18x18 mult.

not those for the I/O ports that are operated at 3.3 V since this I/O is not considered in the CRC model. The resulting energy consumption to process one sample is roughly the same for the FPGA and the multi-context implementation on the CRC model. For the super-pipelined implementation on the CRC model the energy consumption is only about 1/3 compared to the FPGA. For these results, no power optimization techniques were applied. Our work in that area is presented in [14].

## 5. CONCLUSIONS AND FURTHER WORK

Our evaluation of ray casting on processor-like reconfigurable architectures shows considerable improvements in terms of performance, power dissipation, and energy consumption compared to an FPGA. Since the proposed instances of the CRC model are only first prototypes, we expect even better results for refined architectures. Especially the interconnection network can be improved, because a lot of its resources in the current implementation are unused. This, in turn, could also reduce the size of the context memory, which largely contributes to chip area and power dissipation.

Even though a direct comparison to FPGAs with respect to area requirements is not possible, we have also shown that by using processor-like reconfiguration, it is easy to trade off performance vs. area.

## 6. ACKNOWLEDGMENTS

This work is funded by DFG within the ‘‘Priority Program 1148’’ which is focused on reconfigurable computing systems.

## 7. REFERENCES

- [1] R. A. Drebin, L. Carpenter, and P. Hanrahan, ‘‘Volume Rendering,’’ *Computer Graphics*, vol. 22, no. 4, 1988.
- [2] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler, ‘‘The VolumePro Real-Time Ray-Casting System,’’ in *Proceedings of ACM SIGGRAPH*, 1999.
- [3] K. Engel, M. Kraus, and T. Ertl, ‘‘High-quality pre-integrated volume rendering using hardware-accelerated pixel shading,’’ in *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 2001.
- [4] D. Ebert and P. Rheingans, ‘‘Volume Illustration: Non-Photorealistic Rendering of Volume Models,’’ in *Proc. of IEEE Visualization*, 2000.
- [5] G. Knittel, ‘‘The ULTRAVIS System,’’ in *Proceedings of the IEEE Symposium on Volume Visualization*, 2000.
- [6] M. Meißner, U. Kanus, G. Wetekam, J. Hirche, A. Ehlert, W. Straßer, M. Doggett, P. Forthmann, and R. Proksa, ‘‘VIZARD II: A Reconfigurable Interactive Volume Rendering System,’’ in *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 2002.
- [7] M. Motomura, ‘‘A dynamically reconfigurable processor architecture,’’ in *Microprocessor Forum (MPF)*, 2002.
- [8] V. Baumgarte, F. May, A. Nüchel, M. Vorbach, and M. Weinhardt, ‘‘PACT XPP - a self-reconfigurable data processing architecture,’’ in *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2001.
- [9] I. Chen, A. Chun, E. Tsui, H. Honary, and V. Tsai, ‘‘Overview of the intel reconfigurable communications architecture,’’ in *Workshop on Application Specific Processors (WASP)*, 2004.
- [10] T. Oppold, T. Schweizer, T. Kuhn, and W. Rosenstiel, ‘‘A design environment for processor-like reconfigurable hardware,’’ in *IEEE International Conference on Parallel Computing in Electrical Engineering (PARELEC)*, 2004.
- [11] B. T. Phong, ‘‘Illumination for computer generated pictures,’’ *Communications of the ACM*, vol. 6, no. 18, 1975.
- [12] T. Oppold, T. Schweizer, T. Kuhn, and W. Rosenstiel, ‘‘Cost functions for the design of dynamically reconfigurable processor architectures,’’ in *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, 2004.
- [13] U. Kanus, G. Wetekam, and J. Hirche, ‘‘VoxelCache: A Cache-Based Memory Architecture for Volume Graphics,’’ in *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 2003.
- [14] T. Schweizer, J. Oliveira Filho, T. Oppold, T. Kuhn, and W. Rosenstiel, ‘‘Evaluation of temporal-spatial voltage scaling for processor-like reconfigurable architectures,’’ in *Euro DesignCon*, 2005, to appear.