Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 832531, 17 pages
doi:10.1155/2012/832531

Research Article

HoneyComb: An Application-Driven Online Adaptive Reconfigurable Hardware Architecture

Alexander Thomas, Michael Rückauer, and Jürgen Becker

Institut für Technik der Informationsverarbeitung, Karlsruher Institut für Technologie (KIT), Engesserstraße 5, 76131 Karlsruhe, Germany

Correspondence should be addressed to Michael Rückauer, michael.rueckauer@kit.edu

Received 21 February 2012; Accepted 24 May 2012

Academic Editor: Elmar Melcher

Since the introduction of the first reconfigurable devices in 1985 the field of reconfigurable computing developed a broad variety of architectures from fine-grained to coarse-grained types. However, the main disadvantages of the reconfigurable approaches, the costs in area, and power consumption, are still present. This contribution presents a solution for application-driven adaptation of our reconfigurable architecture at register transfer level (RTL) to reduce the resource requirements and power consumption while keeping the flexibility and performance for a predefined set of applications. Furthermore, implemented runtime adaptive features like online routing and configuration sequencing will be presented and discussed. A presentation of the prototype chip of this architecture designed in 90 nm standard cell technology manufactured by TSMC will conclude this contribution.

1. Introduction

Reconfigurable architectures aim to reach the performance and energy-efficiency of application-specific integrated circuits while the flexibility is increased, therefore closing the gap between ASICs and general-purpose processors.

For data-oriented applications an increase in performance compared to general-purpose processors can be reached by mapping operations to a possibly large set of functional units, which are working in parallel. In contrast to ASICs, their actual function and the interconnection between the units are not determined during design and manufacturing but may be changed at runtime to support a wider range of applications.

For example, in a mesh-based architecture, a flexible communication network connects the functional units (FUs) on demand. Since the FUs are communicating directly by exchanging the intermediate results through the communication network, memory accesses for temporary data storage are avoided and memory bandwidth usage is reduced to a minimum. The overall data throughput is at maximum and very close to the ideal performance that can be reached by ASIC implementations.

However, this approach is not without limitations. The increased flexibility comes at the cost of additional hardware. The flexible communication network for FUs requires a lot of multiplexers, communication lines, configuration registers, and additional logic to control the configuration mechanisms. Depending on the type of the reconfigurable approach (coarse-grained or fine-grained) the overhead of the configuration registers and control logic can be considerable. An example for this fact is given by field-programmable gate arrays (FPGAs) [1, 2], which require a lot of configuration data (in the area of several MBs) for specifying the device function. However, FPGAs with their fine-grained approach mark the worst case for this problem. Coarse-grained architectures [3–5] reduce the amount of configuration data to a fraction of the FPGA requirements. This is achieved by vector-based routing and simplified FUs that support arithmetic operations instead of LUT-based Boolean logic. However, the programming models of such architectures are still limited either to native languages or

subsets of known paradigms like C/C++, which prevents the commercial success additionally.

When different application domains, which may have different computational and communication requirements, should be supported, it is not reasonable to integrate FUs with an arbitrary set of operations into a configurable architecture. Therefore, an architecture template, which allows to adapt the set of operations supported by each FU as well as the communication network connecting the FUs at design time, aids in implementing hardware that is even more energy efficient while maintaining flexibility were it is beneficial.

This contribution presents the HoneyComb architecture, a coarse-grained reconfigurable hardware architecture (CGRA) template. It is designed to compute stream-based as well as control-based applications. Therefore, in addition to coarse-grained components, the architecture includes fine-grained components, which are used to compute Boolean operations and control program execution.

To support design-time adaption of the architecture to application requirements in addition to the dynamic reconfiguration, compiler-supported application-tailored hardware reduction/RTL adaption techniques have been developed and implemented.

The following section describes related work. Section three describes the HoneyComb (HC) architecture, its structural characteristics and functions shortly. Section four is devoted to the parameterizable RTL-model and the application-tailored reduction methodology. In section five the RTL-dependent programming model is described. The details of the final prototype and Printed Circuit Board (PCB) design are presented in section six. In section seven we show how the application kernels are mapped onto the architecture. Results and conclusion sections close this contribution.

2. Related Work

In the past decades a number of architectures were proposed in the field of reconfigurable systems that aim to efficiently solve computationally intensive problems while being flexible enough to support a wide range of applications. All these architectures have their advantages and disadvantages. In the following we give an overview of existing architectures.

The Pleiades project [6] proposes an architecture template for ultralow-power high-performance reconfigurable computing. It includes a general purpose processor coupled with a heterogeneous array of autonomous application-specific satellite processors. The resulting assembly is strongly catered to the target application, resulting in limited support for other applications. A well-known example for the use of the Pleiades template is the Maia Chip, which is catered to the requirements of speech coding.

The ACM architecture [7] from QuickSilver Technology is a low-power architecture designed for use in mobile devices. The architecture is derived from an analysis of target applications, resulting in a "fractal" architecture. Heterogeneous nodes are hierarchically connected in a tree fashion with each nonleaf node laid out like the previous layer. Nodes consist of either fine-grained or coarse-grained functional units. The hierarchical assembly facilitates fast reconfiguration of functional units. However, the scalability of the tree-like communication network seems to be a limiting feature in bigger configurations.

The concept of PACT XPP-technologies [8] assumes that applications consist of both regular and irregular parts. Therefore it features a regular array of processing units for dataflow-oriented applications as well as a set of supplementary processors (function folding units, FNC) for control-flow-intensive algorithms. Both components are optimized for 16-bit applications and tightly coupled to support high-speed data transfers. Nevertheless, two completely different components in one architecture require a manual partitioning step. Despite the fact that partial reconfiguration is supported, applications used at the same time on this architecture have to be planned in advance.

Another interesting approach is the DRP architecture [9] from Renesas Electronics (formerly NEC). It consists of a homogeneous multicontext array of processing elements (PE), with each PE having an 8-bit ALU and a register file. In addition there is a context memory, which can choose a new configuration in each cycle. The selection of context is done centrally by a sequencer. The sequencer is a finite state machine that changes states depending on the inner state of the array or depending on external control signals. Memory modules located at the array boundaries provide high-bandwidth data storage. Here, partial reconfiguration seems to be very difficult to realize. Furthermore, data transport to and from the array seems to need additional logic.

The Montium architecture [10] implements a processor that works similar to a VLIW processor, but differs considerably in programming. Instead of instructions, Montium processes sequences of preloaded configurations. This is done by five integrated 16-bit ALUs that are able to execute multiple instructions in a single cycle. Ten local memories with 512 entries each ensure that the ALUs are used to capacity. Multiple Montium processors can be integrated in a System-on-Chip as needed. The maximum parallelism given by the five ALUs seems to be also the limiting factor of this architecture.

The PipeRench Architecture [11] is based on a several times implemented pipeline structure. The individual stages are separated by registers and an interconnect network, in a way that data can be interchanged between pipelines. In addition there is a global network that facilitates data transport contrary to the pipeline flow. The actual configuration of the architecture is determined by parameters and can be adapted to specific applications. The number of concurrent pipeline implementations seems to have a huge effect on the resulting complexity of the interconnect networks and the resulting timing.

PADDI [12] is a multiple-instruction multiple-data (MIMD) architecture. Here, several simple processors that process VLIW-like instructions are connected through a switch structure, which allows conflict-free communication between processors. PADDI is a quite simple architecture.

However the partitioning on the available processors to reach full utilization seems to be a not neglectable task and reminds one of current problems with programming of multicore processors.

The RaPiD architecture [13] is a linear array of functional units, which is configured like a linear pipeline. It is well suited for irregular applications. However, it has weaknesses when processing block-oriented algorithms.

MorphoSys [14] combines all components needed for execution control and data processing in a single design. Execution control is done by a TinyRISC processor, while data processing is performed by an array of processing elements. Array nodes are connected through a multilevel interconnect network. Local memories that hold configuration and data deliver all information needed for program execution and at the same time decouple the array from the host interface.

MATRIX [15] is an architecture similar to MorphoSys, but does not feature an integrated processor for array control. Therefore array control is not as comfortable as with MorphoSys. Data processing is performed with 8-bit precision. MorphoSys and MATRIX both are not able to support concurrent application executions if not planned in advance. This shortcoming is common to the most of the presented architectures, except the ACM.

REMARC [16] is an array based on simple 16-bit nanoprocessors that communicate through local connections. The array is controlled by a global control unit. It carries out transport of data and configurations, but does only provide low bandwidth to the host system. The architecture is designed for multimedia applications. However, it does not have an integrated multiplier, which is a considerable weakness.

Besides the ACM and Montium architectures, most architectures are not designed to run in a multitasking environment. If more than one application is supposed to be running on the same hardware, it is required to plan such a scenario in advance or it is simply not possible to share the resources. For this functionality the target architecture requires additional logic to manage the resource sharing. In case of the HoneyComb architecture, this problem is solved with the adaptive online routing. There, resources for a communication stream are reserved at runtime.

3. HoneyComb Architecture

The HC architecture is an adaptable dynamically reconfigurable cell array with a hexagonal cell layout. The underlying RTL-model is highly parameterizable. Except for the basic structure of the architecture the specification of every component within the array can be enabled, disabled, or modified. A detailed description of the architecture is given in [17, 18]. This section gives only a short overview required for understanding the presented concepts.

The HC array is based on structurally similar cells, which consist of a routing unit and a functional module (see Figure 1). The routing units of all cells are connected to their neighbors and compose the communication network, which is meant to establish point-to-point connections (streams) between functional modules. Supported data types are 32-bit coarse-grained words and multigrained vectors of 1 to N bits.

The routing of streams is performed during runtime and is fully realized in hardware. Therefore routing instructions have to be defined and implanted into the source routing unit. Once a routing instruction is received, the routing unit starts the routing process by propagating the routing instruction to the next cell along the path to the destination cell. The implemented routing algorithm is depth-first search in combination with a backtracking algorithm. Each routing unit requires 3 cycles for the routing process and can process one routing request at a time. Once the destination is reached and the stream is established, data can be sent through this point-to-point channel. Each data word is buffered at the input of each cell, which defines the communication latency by the cycle count equivalent to the count of passed cell edges. The adaptable routing techniques are applied to coarse-grained as well as multigrained data-connections. Each transfer is fully synchronized by a handshake protocol that assures data consistency. Application and configuration data share the same communication network, which increases the reconfiguration performance by using multiple reconfiguration streams at once. It is only limited by the number of cells and the external interface bandwidth.

The functional module specifies the type of the HoneyComb (HC) cells and can be defined as I/O module (IOHC), memory module (MEMHC) or datapath module (DPHC). Cells carrying an I/O module include a specialized microcontroller for data transfers in or out of the array. Therefore the IOHC contains an interface to the system bus (AMBA, WISHBONE, or a proprietary interface). IOHCs initiate all processes within the array by transferring routing instructions into associated RUs, establishing data and configuration streams to destination cells, configuring these cells and controlling data transfers to and from these cells. Streams between cells can be routed by tunneling the necessary routing instructions to the source and starting the desired routing process. All transfers can be done through DMA without the interference of a system controller. Therefore an optimized μ Controller has been integrated into the IOHC, which includes parallel working address generators for fast data transfers.

The MEMHCs provide storage space for the applications within the array. Therefore multiple memory modules can be included in each MEMHC and store coarse-grained and multigrained data simultaneously. Logical merging of memory modules can be done to offer bigger storage space for applications. Each memory configuration can be used as RAM as wells as a FIFO or a LIFO.

DPHCs realize the main arithmetic or logical data manipulation units within the array. Therefore their functional modules include ALUs, LUTs, coarse-grained and fine-grained registers, data type converters, and data branchers, which are required to split synchronized data streams. The combination of ALUs and LUTs in one module allows the evaluation of ALU operation flags (carry, sign, overflow, etc.) at once and influence the next operations. Therefore each ALU includes a context memory, which selects

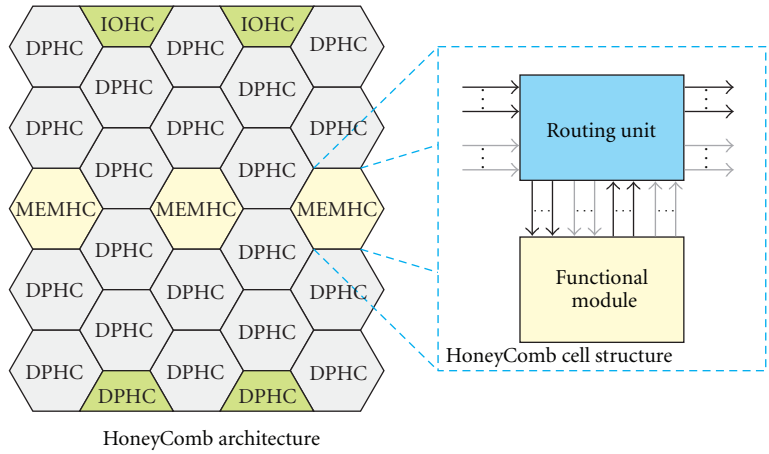


Figure 1: Hexagonal HoneyComb (HC) array with unified cell structure and three cell types—DPHC, MEMHC, and IOHC.

the opcodes, operands, output registers and generated flags. It can be addressed by LUTs or directly by output flags and change the output on every cycle. LUTs and fine-grained registers can implement finite state machines (FSMs) for even more complex functions. By sending fine-grained or coarse-grained output data to IOHCs, the removal of the current configuration can be triggered and a new one started. To increase the bandwidth of the I/O operations IOHCs are provided with a separate clock. Usually this clock is higher compared to the array clock. Array configurations consume the incoming data much faster than a system bus can deliver especially if multiple data streams are required. In this case the ratio of both clocks can be adjusted by the system.

Since every cell is capable to be configured independently partial reconfiguration is possible. Runtime routing even allows overlapping of configurations without being considered at compile time if enough resources are available. Clock gating of idle parts of the architecture is implemented and performed in two levels. The first level controls the routing units of each cell independently. Thereby, a routing unit is activated if the neighbor cell is establishing a path to this cell. Functional modules of MEMHCs and DPHCs are clock-gated if no configurations are programmed. Incoming configurations activate the cells and keep them active until the configurations are deleted. If the user detects a nonfunctional routing unit the affected cell can be deactivated. In this case the HC array routes the routing requests without using the deactivated cell.

The idle HC architecture is started by providing one of the available IOHCs with a memory address where program execution should start. This program includes steps to initiate the configurations, transfer the data to and from the array, and clean up the array by deleting the configurations. Several IOHCs can be used at the same time if the addressed cells within the array are disjunctive. The scheduling and checking of configurations is supposed to be performed by a runtime system, which is not finished yet. Control operations of the array are done by the HC Controller, which offers a separate interface for the system's hosts and a set of registers for checking cell states and writing control values.

The architecture can be programmed using the HC assembly language or the HC language, which are both introduced in the latter sections. The HoneyComb assembly language is a low level language for structural programming while HoneyComb language is a higher level approach.

4. Parameterizable RTL-Model

This section describes the methodology we have used to design the HC architecture and additional tools we have developed to support the configuration and verification process.

4.1. Design Methodology

The HC architecture is a highly parametrizable architecture. The complete model has been developed based on VHDL and its generic capabilities. Almost every possible way of specifying generic structures in VHDL has been used, including:

(i) generics definitions;
(ii) constant definitions;
(iii) conditional and looping generate structures;
(iv) function library for parameter evaluation;
(v) package definitions for configuration management.

The combination of these techniques is a very powerful way to describe parametrizable designs. We have defined two main sets of configuration parameters. The first set is defined within the global constant packages and includes constant definitions considering global parameters like

(i) array size;
(ii) CG/MG data width;
(iii) routing instructions formats;
(iv) IOHCs instruction formats;
(v) clock gating enable/disable.

The second part describes local definitions regarding

(i) cell port count;

(ii) DPHC configurations;

(iii) MEMHC configurations;

(iv) IOHC configurations.

Depending on the given parameter sets, specific features can be activated or module instances removed.

To ease the debugging process careful signal definitions have been implemented. With a few exceptions there are no unused signals available. This way every structural problem causes undefined signal states and can be identified very fast. Additionally, all modules have been designed with intention of exhaustive reuse ability. So, if fixed in a specific module this change is automatically applied to all instances of this module in the hierarchy. The more modules of the same type are instantiated the bigger is the impact.

Following these simple rules we designed the whole architecture in VHDL in about two years. The structural correctness and the functional verification took additional two weeks. After this time the architecture was able to perform first tests and simple applications for verification purposes. It took additional 4 years for the development of the programming languages, debugging tools, and demonstration application (see later sections). The final step was the IC layout in 90 nm for the prototype.

4.2. Configuration Manager. The amount of RTL configuration parameters is enormous. The biggest part has been spent for specifying the functional modules within the DPHCs, which includes over 60.000 parameters. Parameters can specify the operation sets, the count or available modes (single context/multicontext) of each single ALU, LUT configurations, the number and type of registers, the interconnection between these modules, and so on. Manual control of this amount of parameters is simply not possible and requires additional tool support. Therefore, the configuration manager has been developed.

The main purpose of this tool is the management of configurations. However, the functionality is going a step further and includes the generation and merging of configurations as well. The generation of configurations can be done manually or by analyzing previously compiled applications (see Figure 2).

Based on predefined templates or ideal array representations the analysis of applications is performed by the HCL compiler (see Section 4). The result is a functional description of the application in assembly language. One or several of those descriptions can be used to extract the necessary specifications for the target RTL description of the array, so-called super RTL configuration. Since configuration code for DPHCs and MEMHCs is structural, it is quite simple to extract the necessary information. Figure 3 shows how several applications impact the functional units. This kind of merging is performed on all levels of functional units and results in a hardware structure that is able to support every considered application.

The configuration manager has been developed in VBA for Microsoft Excel. The table management of this application is very well suited for configuration management and allows manual configuration creation. With additional VBA code necessary consistency checks have been implemented to support manual work. The super configuration Generator is part of this application as well as the Assembler application, which is described in Section 4.

5. Programming Model and Tools

The structural composition of the HC architecture predefines the programming model. This model is composed of three layers according to Figure 4 and is executed by specific architecture parts. Transport layer is performed by IOHC and is meant for conditional or unconditional data transfers. The communication layer controls the routing network and influences the placement of configurations. The configuration layer describes the functional modules of MEMHCs and DPHCs.

The programming model impacts the definitions of the HC Language as well as the definition of the HC assembly language; both are described in the following subsections.

5.1. HoneyComb Assembler (HCA). The HoneyComb Assembler application is integrated into the configuration manager. Thus, all configuration parameters of the array are available to the assembler and are considered during assembly. The HCA language definition consists of three parts, one for each layer of the programming model. By specifying the target cell coordinates and the layer type the user tells the assembler which kind of code to generate. The code specification for the transport and communication layers is globally the same for the given array. In case of the configuration layer the resulting binary code characteristics can differ from cell to cell if the RTL configurations vary, that is, if the HC array is heterogeneous. Code adapting to RTL configuration helps to reduce the resulting binary code size and required hardware structures but makes it incompatible to other cells.

Since the multicontext capability of HCA can be disabled at RTL it is required to perform code transformations during the assembly process. The following example demonstrates the necessity for transformations:

```
ALUOP   0, ADD
```

This instruction programs the given ALU to perform the ADD operation. In case of the ALU with multicontext capability this instruction has to be translated to the following piece of code:

```
ALULCFG   0, C0=[ADD]    #context 0
ALULCFG   0, C1=[ADD]    #context 1
...
```

Here, every active context will perform the ADD operation what in fact results in the same behavior as the ALU without multicontext capabilities. The assembly application transforms this kind of transformations automatically. Therefore,

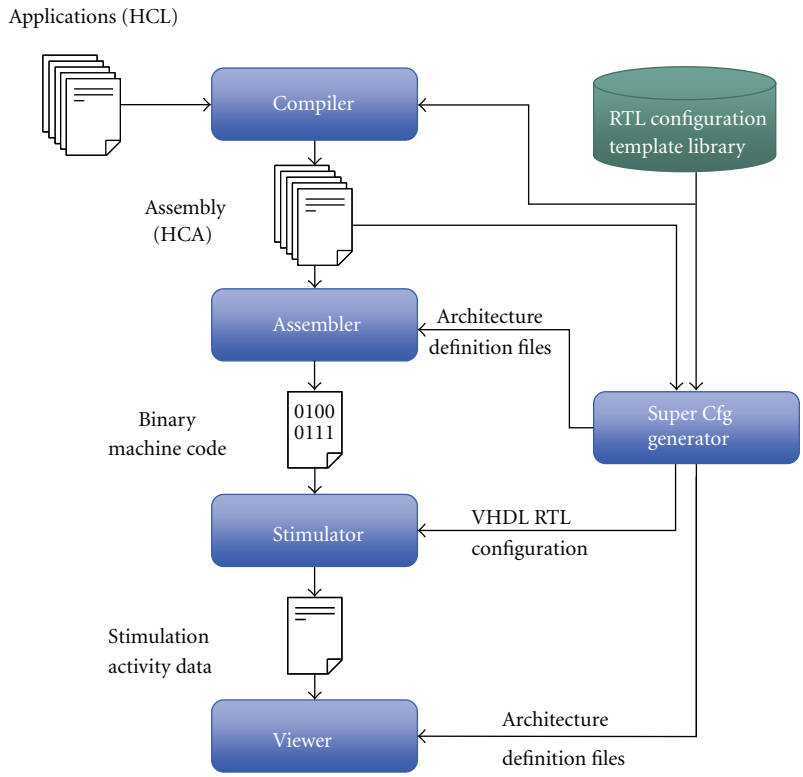


Figure 2: HoneyComb application-tailored design flow for generation of RTL configurations, development, and debugging of applications.

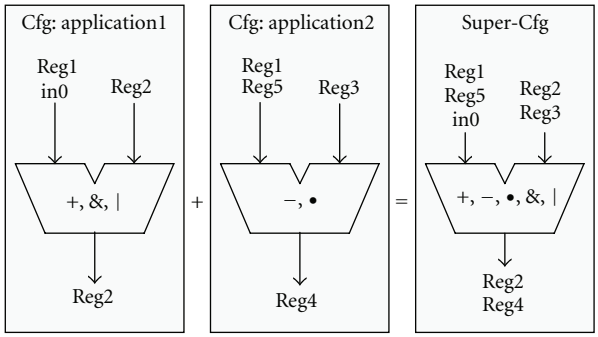


Figure 3: Example for ALU configuration generation derived from two applications and merged to the final operation and register sets.

the programmer does not have to consider every detail of the architecture and would still be able to compile applications. This kind of transformations performs the assembly tool in the extended mode which has to be separately activated. If incompatibilities are detected, the user will be notified and can adapt his code manually.

5.2. HoneyComb Language (HCL). The HoneyComb assembly language describes the configuration of a HoneyComb array in both a low level and structural fashion, making it hard to describe the desired behavior. Therefore we introduced the HoneyComb language (HCL), a high-level language that makes it much simpler to describe the behavior of the three cell types. In addition it is possible to specify sub-configurations consisting of several cells that together perform a more complex function than is possible with a single cell.

A structural programming language is used to describe the behavior of an IO cell. It features instructions for configuration management, off-chip communication, and streaming data transfers between the IO cell and the array cells. Using subconfiguration descriptions, configurations consisting of several cells can be established and removed with a single statement.

The behavior of a data path cell is described as a sequential process executed once per clock cycle. It supports control flow statements and assignment statements that assign the result of a complex expression to a register or output port.

5.3. HCL Compiler. A program written in the HoneyComb language is transformed into HoneyComb assembly language with the HCL Compiler software.

In the first step of compilation the HCL description is read in and transformed into an abstract syntax tree (AST) representation. The further proceeding depends on the type of cell description.

The AST of an IO cell description is transformed into an intermediate representation where all source language statements are replaced by sequences of one or more assembly instructions. Control flow statements are transformed into equivalent structures utilizing only conditional jumps.

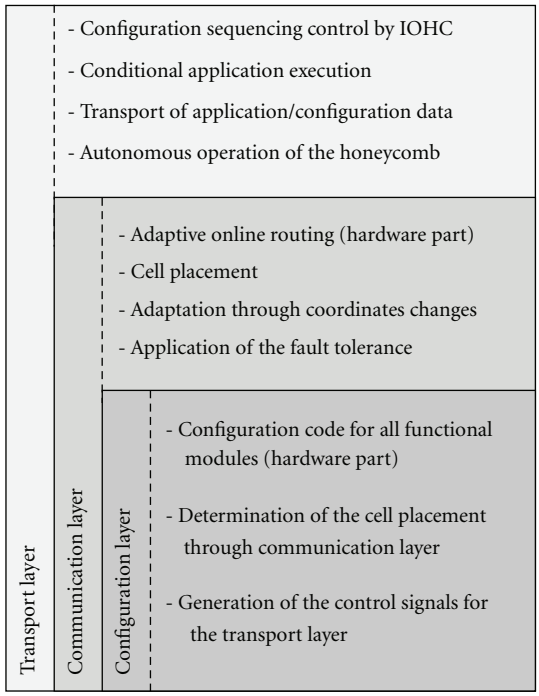


Figure 4: Programming model for the HoneyComb architecture based on the structural hierarchy.

This yields a control flow graph consisting of instruction base blocks that are connected by edges where a control flow transfer from one block to another may occur. There is no dedicated instruction selection and scheduling except for complex expressions, for which instructions are ordered to minimize the number of registers needed to hold intermediate values. Register allocation is done with the graph coloring approach described in [19]. In assembly code, basic blocks connected by unconditional control flow edges are emitted consecutively if possible to reduce the number of control flow transfers.

When the compiler processes a sequential description of a data path cell's behavior it first puts all noncontrol flow statements into a list. For each statement in the list, its actual execution condition is calculated by looking at the conditional statements surrounding it and the execution conditions of statements writing to the same variables. Consider the following example:

$$\text{IF } a \text{ THEN IF } b \text{ THEN } x \leq y + z; \text{ END IF; END IF;} \quad (1)$$

$$\text{IF } c \text{ THEN } x \leq z - y; \text{ END IF;} \quad (2)$$

The execution condition of the first statement is (a and b and not c) as it is only executed when the conditions of the both surrounding if statements are true (a and b) and the execution condition of the second statement (c) is false (equivalent to not c = true). Because of the synchronous nature of the HoneyComb architecture we also need to record the condition under which the value of a variable is consumed. Again consider the above example: the values of a and c are always consumed, while the value of b is only consumed if a is true. The values of y and z are only consumed if either statement one or two is executed. This information is collected for each statement in the list. With this information the statements in the list can be executed in any order assuming that any writes will be visible in the next cycle.

The statements in the list are now mapped to the ALUs and LUTs of the cell. Variables used in the statements are mapped to registers and IO ports of the cell depending on their type. In order to minimize resource usage, operations with mutual exclusive execution conditions and matching evaluation conditions for common variables are mapped to the same function unit.

This is done using a clique-partitioning algorithm [20] that is employed twice, once for the mapping to the ALUs and once for the mapping to the LUTs. A graph is created whose vertices represent the statements to be mapped to the functional units. Two vertices are connected by an edge if they can be mapped to the same functional unit. The edges are weighted by the cost reduction that is achieved by merging the two adjacent vertices. In each iteration the edge with the highest weight is selected and its adjacent nodes are merged into one. Nodes that were adjacent to both merged nodes are connected to the resulting node. This merge process is repeated until no edges are left or all remaining edges have a cost increase associated with them. Each resulting node is mapped to one ALU or LUT respectively.

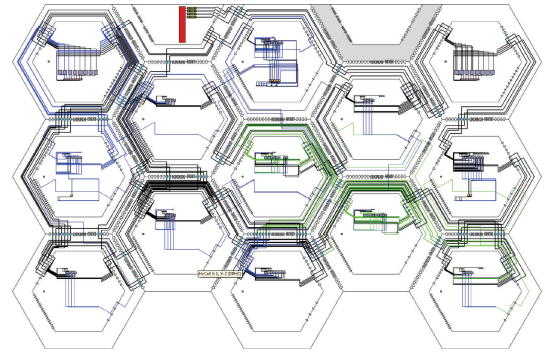


Figure 5: Execution of the AES256 algorithm executing on the HoneyComb architecture visualized by the HCViewer.

5.4. HoneyComb Viewer (HCViewer). The last step in the software development for the HC architecture is the debugging using the HCViewer tool (see Figure 5). This tool requires a current HC configuration generated by the Super Configuration Manager and the simulation activity data, which can be generated by the HC VHDL model during execution of an application. Once loaded, the HCViewer visualizes the processes within the array and reports all user relevant data and values.

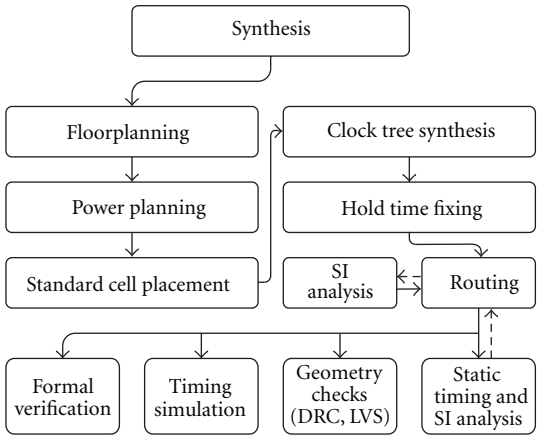


Figure 6: Implementation and Sign-Off Flow.

6. Prototype IC

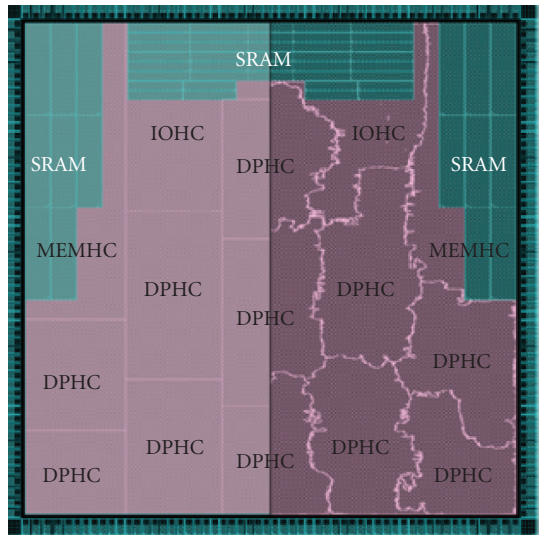
6.1. Initial Technology Decisions. We had to make the choice between different IC manufacturers providing a 90 nm process. As area was our main concern, we chose TSMC, which had the process with the best area efficiency. We selected a low-power library, which trades performance for a higher transistor density. Therefore the test chip was expected to achieve a lower clock frequency than a production chip would.

Initial synthesis of the array's RTL model was done with the design compiler software from Synopsys. The resulting netlist was imported into the SoC Encounter software from Cadence, where the entire layout work took place. Figure 6 gives an overview over the implementation and verification flow, which is described in detail in the next two sections.

6.2. Layout. From the total die area of 16 mm^2, a border of 183 μm had to be reserved around the standard cell area for the seal ring, the IO and bond pad area and the core power ring. This left an area of 13.2 mm^2 for the actual design. We chose a flat design methodology with a single consecutive workflow. This saved the overhead of having to implement and characterize the submodules separately.

Figure 7(a) shows the layout of the cells and the placement of the SRAM modules on the actual chip. Unlike the hexagonal cells in the logical layout, the physical cells have a rectangular shape. To connect each cell to its six neighbors despite the rectangular shape, the arrangement of the cells is staggered, so that an inner cell still adjoins its six neighbors.

Figure 8 shows the chip's power network. It consists of a power ring around the chip's core area and a regular power grid that spans the entire core area. With the chosen technology, nine layers of metal are available for routing. The two topmost layers are approximately three times thicker than the others, giving them a lower resistance and a higher current tolerance compared to the others. Therefore, they were chosen for carrying the power grid. The geometry of the power grid was chosen based on a suggestion found



(a) (b)

Figure 7: Cell layout as defined by floorplanning (a) and after standard cell placement, respectively (b).

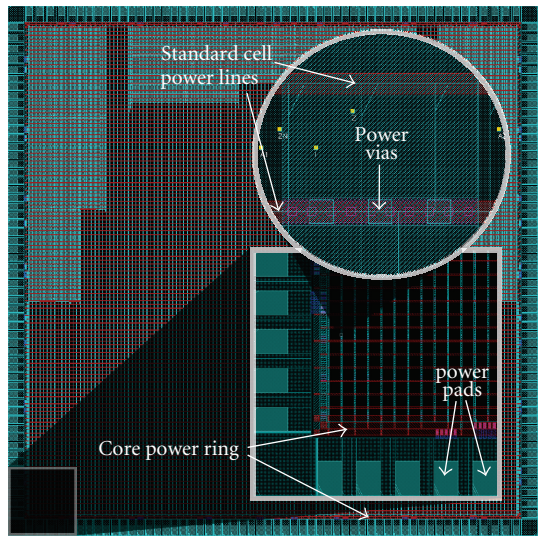


Figure 8: Power network.

in the application note for the employed standard cell library. An estimation of the chip's power consumption was made with the VoltageStorm power analyzer. That value was doubled to obtain a comfortable safety margin and the grid was planned to meet these requirements.

We added as many power pads as allowed by the spacing rules of the IO pads: 32 pads carrying power and ground for the IO domain and 58 pads carrying power and ground for the internal power supply, which is well above the requirements.

Afterwards the standard cells were placed. Initially the placer tended to create crowded areas with local placement densities of nearly 100%, which made it impossible to add buffers during hold time fixing or to obtain a valid routing

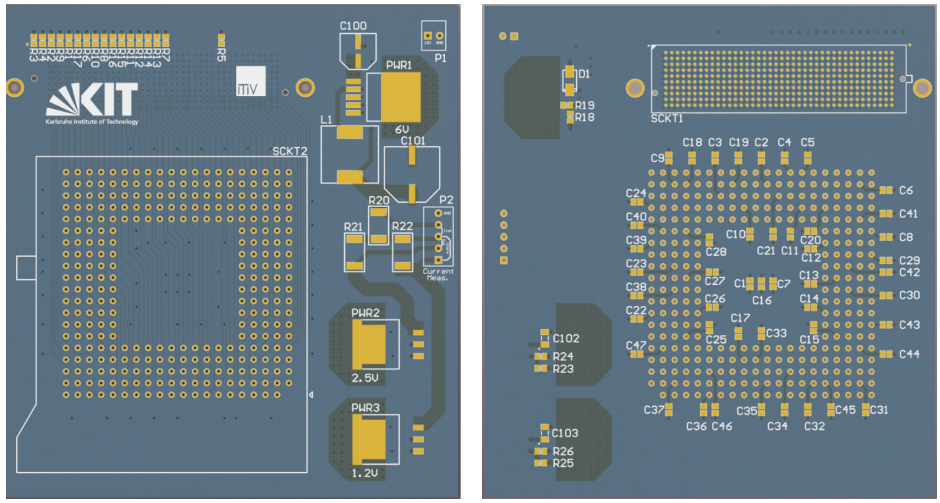


Figure 9: 3D rendering of final HoneyComb board layout.

of signal nets later on. Therefore the placement flow was altered. After an initial placement multiple iterations of timing optimization and incremental placement were performed. Thereby the allowable local placement density was raised from an initial 70% up to 80%. This approach created a more uniform standard cell distribution and leads to better timing results. Figure 7(b) shows the final cell shapes after placement. They are irregular and blend into each other, which is beneficial for timing.

Clock tree synthesis was done with the automatic synthesis mode, where the software created the clock tree geometry automatically from few parameters, which are the maximum insertion delay and the maximum skew between two flip flops' clock signals. With the default wire geometry the current through the clock tree exceeded the current limit. Therefore a custom rule was used that tripled the wire width between the inner clock tree buffers. After clock tree synthesis, another timing optimization step was performed that analyzed and repaired any remaining hold time violations by inserting buffers into affected signal paths.

Up until this point all work was done on a partially routed design likely to contain unrouted nets and shorts between different nets. A timing-driven routing algorithm legalized this routing, while trying to minimize the timing impact caused by the wiring.

Finally a signal integrity analysis was performed on the fully routed and timing-clean design. Hereby, pairs of nets that could influence each other through capacitive coupling were identified. If this effect could have caused a net to carry an invalid value, it was rerouted to reduce the coupling. This analysis-and-repair step was repeated until the design was free of signal integrity errors.

6.3. Chip Sign Off. Timing and signal integrity were checked with PrimeTime SI, using a 10% derating of clock and signal path runtimes to account for timing uncertainties introduced by manufacturing tolerances and different voltage levels across the chip. First, the software detected a few hundred transition time and signal integrity violations. A script extracted the violations from the timing reports and created another script that repaired the errors within encounter. The transition time violations were fixed by upsizing drivers; the signal-integrity violations were fixed by rerouting the affected nets. It actually took a lot of iterations and more than a week to fix all of these violations.

In addition, design rule checks (DRCs) and layout versus schematic (LVS) checks were performed with the Calibre software. As we did not have access to the actual layout data of the memories and standard cells, they had to be treated as black boxes during LVS. Therefore only the connectivity of the cells could be checked.

To verify the functional equality of the synthesis netlist and the netlist representing the final layout, the netlists were compared with the formality software. The software found three discrepancies between the synthesis netlist and the netlist representing the final layout. Despite this, we verified that the postlayout circuit performs the same function as the original one.

Before the layout was sent to manufacturing we performed a successful simulation run with timing data from Primetime SI.

6.4. Printed Circuit Board Design. The HoneyComb prototype IC has a proprietary parallel interface that is optimized to maximize throughput of streaming data transfers. It has two 32-bit data interfaces, of which one is dedicated to data input while the other is dedicated to data output. At 125 MHz this results in a combined bandwidth of 1000 MB/s, 500 MB/s in each direction.

The prototype IC is supposed to be connected to an FPGA, which serves as an interface between the HoneyComb IC and the controlling host system as well as an external memory controller that connects the IC to external DRAM memory. This allows for easy evaluation of different interfacing options as well as slave (accelerator) and standalone mode.

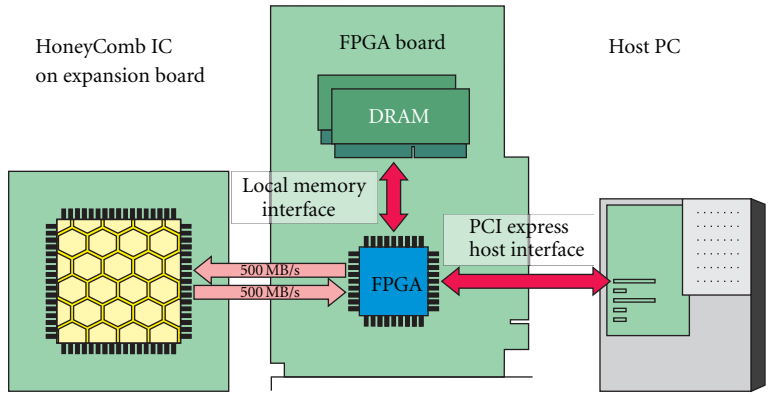


Figure 10: HoneyComb IC in coprocessor configuration with PCI-express host interface.

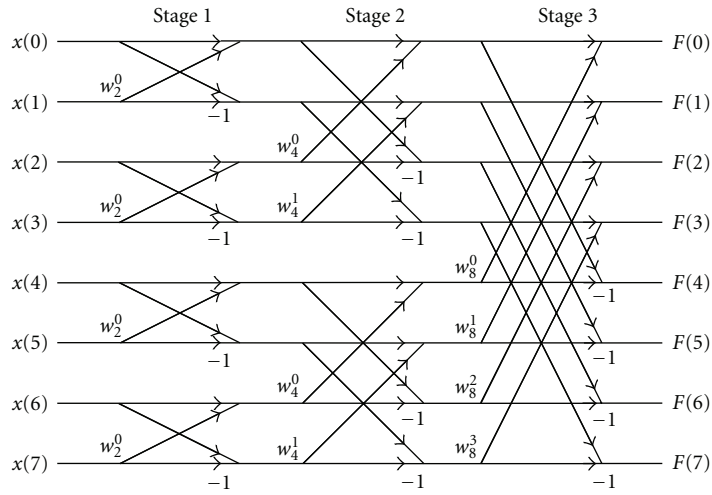


Figure 11: Radix-2 butterfly structure for 8-point FFT implementation.

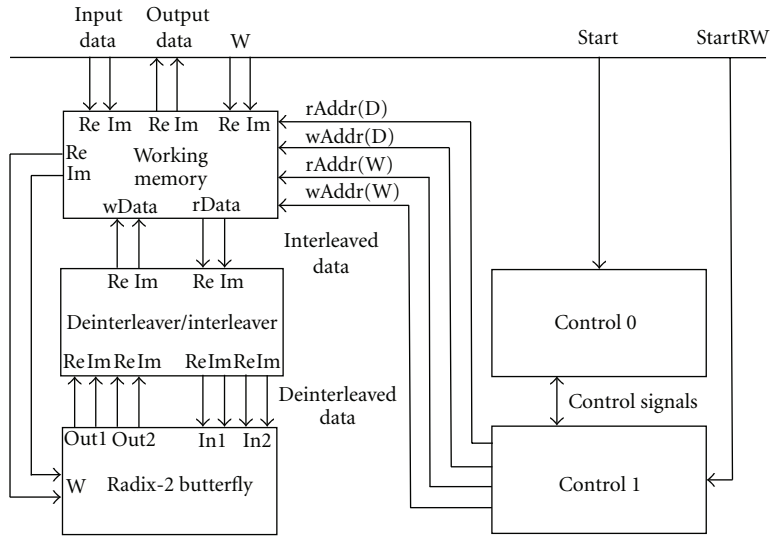


Figure 12: HoneyComb implementation of the FFT algorithm based on radix-2 butterfly approach.

The test board for the HoneyComb prototype is designed as an FPGA Mezzanine Card (FMC) as defined by the VITA 57 standard [21], which is an expansion card standard supported by recent Xilinx FPGA evaluation boards. The HoneyComb IC has 134 signal connections to the FPGA, therefore the board features a 400 pin high pin count (HPC) connectors, which supports up to 160 single-ended I/Os. It can be used with board featuring an FMC HPC connector like the Virtex-6 ML605, Kintex-7 KC705, or Virtex-7 VC707 evaluation kits. A 3D render image of the final layout of the HoneyComb board, which is currently in production, is shown in Figure 9.

In any of these configurations, the HoneyComb IC can be used as an application accelerator within a Standard-PC, to which it is connected via the PCI-Express slot of the base board. In this configuration, which is shown in Figure 10, the HoneyComb IC has access to the memory located on the FPGA board as well as to the PC's main memory. Configuration and computation are initiated by the host-PC.

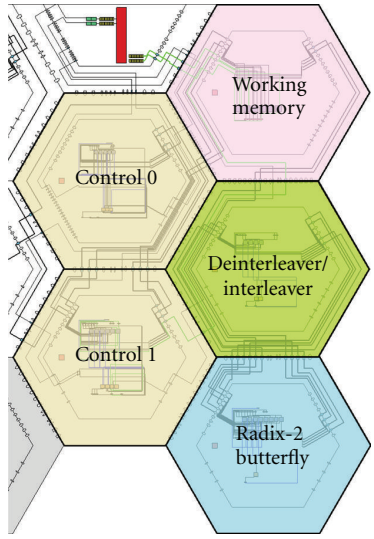


Figure 13: Mapping of the radix-2 FFT algorithm on the HoneyComb architecture.

7. Mapping Applications onto the HoneyComb Architecture

For demonstration purposes we developed a set of applications, which includes a 1024-point FFT [22], Wavelet algorithm according to JPEG2000 specification [23], iMDCT [24], and Advanced Encryption Standard (AES256) [25] implementation. These applications have been mapped on the HC array and the calculated data sets have been compared with reference implementations in C/C++ with a perfect match. Since both implementations are working with 32 bit precision these results were possible and expected. As described in the layout section, we applied the application-tailored reduction technique using these applications to reduce the resource requirements of the prototype in order to fit the design into an area of 16 mm^2.

The following procedure was used to generate the executable program code and to adapt the HoneyComb model to application requirements. We implement the initial high level program in HCL based on the C/C++ reference implementation. After compiling the HCL description we received the HCA description, which is a low level assembly program for the HoneyComb array. After additional optimizations we used the assembly code to determine the target HoneyComb model by adding additional resources to the architecture according to FFT requirements. Finally, the HC-assembler has been used to generate the final binary program code which can be executed by the modified HoneyComb model.

The following subsections will give an overview how the target applications have been mapped onto our architecture.

7.1. 1024-Point FFT Algorithm. The fast Fourier transform [22] is an efficient algorithm (Cooley-Tukey [26]) for discrete Fourier transform (DFT). It computes frequency components for a given sequence of values which can be a number of consecutive samples for a given signal. Besides the digital signal processing this algorithm is important for a wide variety of applications.

While direct DFT calculation results in complexity of $O(N^2)$ the FFT reduces it to O(NlogN) and allows calculation of datasets with thousands of point in a relative short time. The usual hardware implementation is based on butterfly structures. Each butterfly has a specified number of inputs and outputs which is specified as radix-parameter. In case of two inputs we are talking about radix-2 implementation. Figure 11 shows the basic structure for a radix-2 butter fly structure with eight points.

For the mapping of this algorithm we chose the radix-2 implementation. Since the resulting array composition has not enough resources to implement the complete radix-2 FFT directly we decided to map this algorithm partly in the area and partly into the time domain. This approach is supported by the HoneyComb array by using the multicontext capability. Therefore the working data has to be stored in memory modules (MEMHCs) and retrieved according to the butterfly structure. To generate the required address sequences two separate HoneyComb cells are dedicated to this task and are marked as control cells (see Figure 12). One separate MEMHC has been assigned for holding the coefficients W_i and current working data. Depending on incoming addresses the memory cell retrieves addressed data and coefficients. Since current MEMHCs only include single port memories retrieved data is read one by one on each cycle. An additional deinterleaver cell is used to parallelize sequential data values and forward each pair of values (In1, In2) to the butterfly cell. Thus, the butterfly cell is able to calculate one output pair (Out1, Out2) every two cycles and forward the results through the interleaver to the memory cell.

Two external signals control the execution of the FFT algorithm: StartRW and Start. StartRW controls whether the working data and coefficients are read from or written

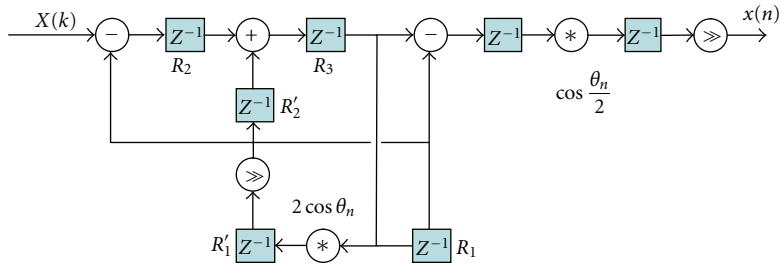


Figure 14: iMDCT structure fitting hardware implementation.

to the working memory. It triggers a linear address generator which generates an address stream on predefined output port of the control 1 cell. Whether working data or coefficients are transfers depends on the connected memory interface. These connections are established by dynamic reconfiguration when memory has to be filled or calculated data is to be retrieved. Once the memory is loaded the calculation can be started with the Start signal.

The butterfly uses a fixed point representation for the calculation which can be adapted to application requirements by resetting the fixed point position. The resulting algorithm is not limited to 1024-point FFT. Depending on the stored coefficients W_i it is possible to calculate data sets of different sizes. The limiting factor is only defined by the maximum size of the memory modules in the MEMHC. The ASIC implementation includes eight 1024 × 32bit memory modules. So, the maximum size is limited to 2048-point FFTs by this version.

Figure 13 shows one possible FFT configuration on the HoneyComb array as described in the sections above. The whole configuration requires 4 DPHCs and 1 MEMHC.

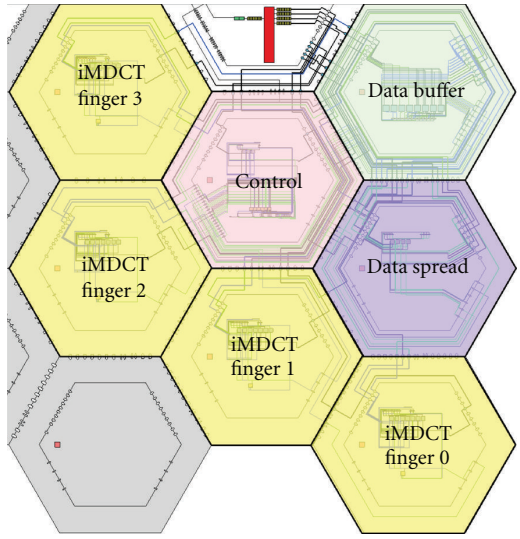


Figure 15: Mapping of the iMDCT algorithm onto the HoneyComb architecture.

7.2. 1024-Point iMDCT Algorithm. The inverse modified discrete cosine transform is an algorithm quite similar to the prior mentioned DFT. This modified version is primarily used for audio compression standards like MP3, AAC, and OggVorbis. The direct calculation of as iMDCT has the same complexity as the DFT which is $O(N^2)$. Like in case of FFT similar butterfly-based approaches exist to reduce the complexity. There are two particularities of this algorithm worth mentioning: for once the fact that it works with real numbers only and for second that a given number N of spectral values results in 2*N samples. The first point halves the memory requirements compared to FFTs since only real numbers have to be stored. In case of 1024-point iMDCT we get 2048 sample values, which is specified according to OggVorbis Audio compression.

Since one butterfly approach has already been implemented on the HoneyComb array we decided to use a more direct approach. Therefore we used the transformations introduced by Nokolajevic and Fettweiss [24] resulting in a structure which can be implemented directly in hardware. Figure 14 shows a slightly modified version after adding a few additional registers.

Even though the algorithm does not pursue the same approach as the FFT, the implementation resulted in a similar configuration. One MEMHC is used to store working data as well as cosine coefficients. But instead of using the MEMHC in RAM mode it is used in FIFO mode. Though, once a value has been read it has to be put back if it is supposed to be available again. This function is performed by a separate cell (data spread), which distributes the values from memories and forwards a copy to the iMDCT finger cells. Those cells implement the complete iMDCT structure as shown in Figure 15.

To compute a single output the iMDCT structure requires to receive 1024 spectral and cosine ($2\cos\theta_n$) values. On the final iteration the multiplication with the cosine value ($\cos(\theta/2)$) finalizes the calculation. Since the algorithm is working with fixed point numbers shift operations are required to correct the results. The resulting values are forwarded through the IOHC cell directly out of the array.

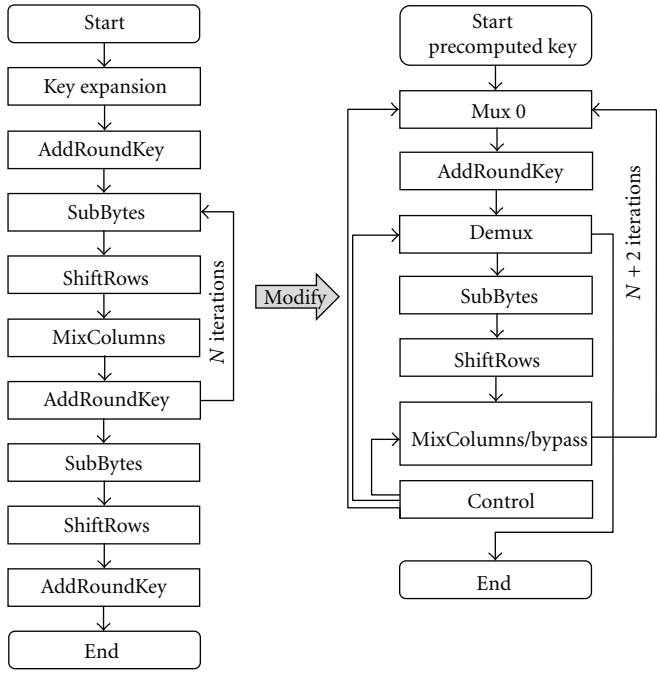


Figure 16: Original AES operation scheme and the modification of the implementation for the HC-Array.

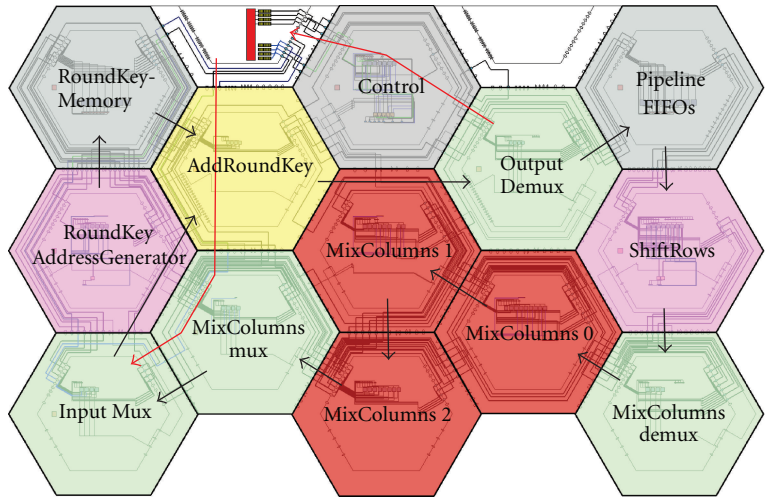


Figure 17: Resulting mapping if the AES algorithm on the HoneyComb architecture.

This approach is quite slow compared to butterfly solution. However, it is a good example to demonstrate multi-context capabilities of the DPHCs.

7.3. The Advanced Encryption Standard. The Advanced Encryption Standard (AES) [25] is a symmetric-key encryption algorithm standardized by NIST [27] in 2000. The symmetric character of the algorithm allows encryption and decryption of electronic data with the same cypher key. The algorithm is fast in software as well as in hardware. However, due to its nature the latter is always more efficient.

AES is using a block code of 4×4 bytes blocks. A set of transformation operations are defined which are executed in repetitions in several rounds to perform the encryption of the input text. Additionally, reverse rounds are defined to reverse the encryption with the same encryption key.

Four high level steps are defined for the encryption rounds: AddRoundKey, SubBytes, ShiftRows and MixColumns. AddRoundKey combines current block elements with the cypher key by applying the xor operation. SubBytes substitutes the bytes within the blocks due to a given lookup table. ShiftRows reorders the rows of the given block, while the MixColumns function reorders the columns of the block.

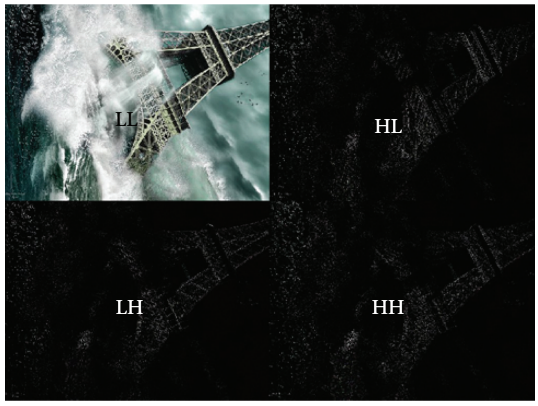


Figure 18: 2D wavelet transformed image.

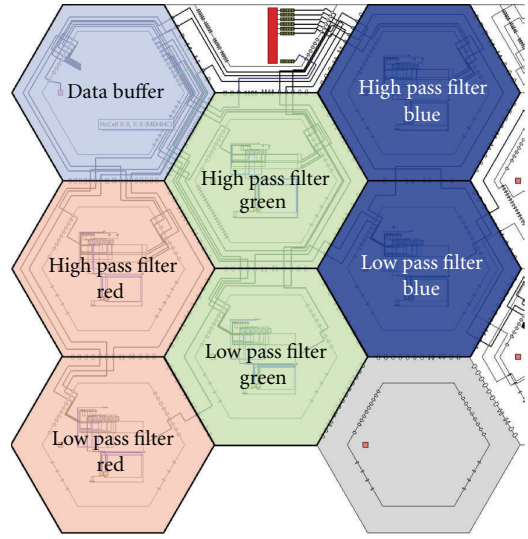


Figure 19: 3-time wavelet algorithm implementation on the HoneyComb architecture.

AES key sizes are specified with 128, 192, and 256 bits. Before starting the actual data encryption the plain text key has to be expanded. Here, the key is divided in 128-bit blocks (RoundKeys), whereas the blocks are partly filled with the key data and partly are calculated recursively. After the expansion the RoundKey can be applied with the AddRoundKey operation on the data during the encryption. Since the key expansion is not limiting the AES performance it will be done in software and the RoundKey will be transferred into the HoneyComb array for the actual encryption process.

The AES algorithm specifies a very specific order for the execution of those operations. However, to reduce the resource requirements, the block diagram has been slightly modified. Due to hardware restrictions, it is not possible to call any function at will, so each path has to be defined in advanced, what is shown with each path on the right hand side of Figure 16. It was important to avoid multiple implementations of the same function to save as many resources as possible. Therefore, the whole algorithm is working both ways: in parallel mapped in the hardware area and sequential by iterating the data by reusing the same resources over and over again. Depending on the key size the number of iterations changes, so all key sizes are supported by the HoneyComb implementation.

Figure 17 shows the results mapping of the AES algorithm on the HoneyComb architecture. The darts in this figure represent data paths between operational cells and have always four concurrent connections to transport one column of a block at once. Thus, every four cycle one block is transferred from one cell to another. The MEMHC with the pipeline functionality is used for once to increase the pipeline depth of one round and secondly holds the lookup table data for the SubBytes operation. The increasing of the pipeline depth increases the efficiency of the mapping enormously. Since this algorithm is working iteratively, it is required to empty the pipeline completely before starting the calculation for the new data set. This refill period leads to idle time on the array and decreases the resulting performance. By using deeper pipelines it is possible to reduce the idle time relatively to calculation time.

The result is high performance and efficiency. The maximum pipeline depth given by the FIFOs has 1024 entries, so including the calculation cells with additional register stages over 256 blocks can be calculated in one pass. The final mapping of the AES algorithm requires all available cells on our ASIC: two MEMHCs and eleven DPHCs.

7.4. Wavelet Application. The Wavelet algorithm [23] works on whole images to filter the higher and lower frequencies of the color dispersion. It can be applied horizontally and vertically to achieve 2Dtransformation. The JPEG2000 standard uses this algorithm to separate frequencies for better compression results. Quite similar to the JPEG approach loss of high frequency shares cannot be noted by the eye, so loss of this kind of information does not degrade the reextracted image noticeably.

Simply spoken, the algorithm compares three neighboring pixels and calculates the high and low frequency shares. When this is done horizontally, the results are two half-sized images, whereas the left image contains the low pass results and the right image contains the high pass results. When the algorithm is applied vertically this results in an image with the same size but consisting of four separate quarters (see Figure 18). Each quarter is the result of horizontal and vertical low and high pass filtering, what is marked with the letter L/H. The first letter indicates horizontal filtering, the second the vertical filtering.

The low and high pass filters require each one cell on the HoneyComb architecture. So, to implement a complete wavelet filter we need two DPHC cells to handle one data stream, which can be a color component, like red, green or blue. Because of tight timing constraints between these two cells a FIFO has been used to improve the performance.

To be able to compute all three color components of an RGB image and to improve the resulting performance we implemented three times the complete wavelet filter,

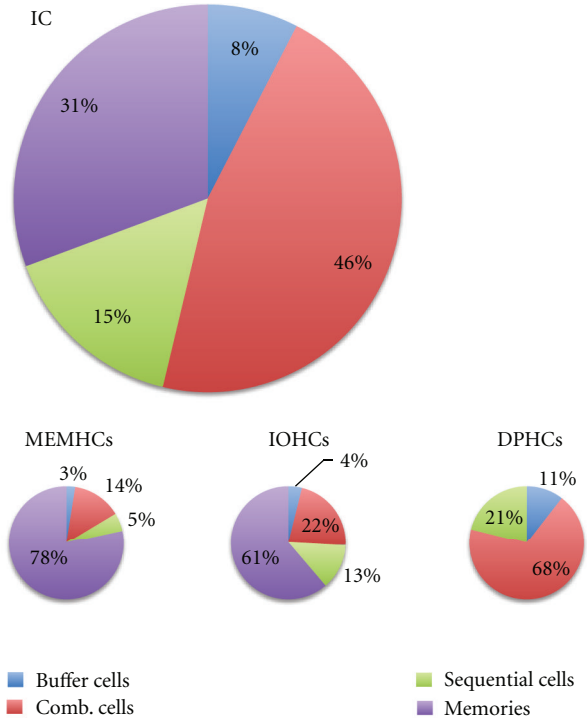


Figure 20: Distribution of the standard cell types within the HoneyComb array prototype.

what resulted in a configuration with six DPHCs and one MEMHC (see Figure 19).

8. Results Overview

The applications described in the last section have been taken for performance evaluation. These applications have been mapped on the HC array and the calculated data sets have been compared with reference implementations in C/C++. As described in the layout section, these applications have been used for the reduction of the prototype to reduce the resource requirements and to fit the design into an area of 16 mm^2.

The application results, which were obtained by cycle-accurate simulation of the architecture's VHDL model and normalized to 100 MHz, can be found in Tables 1 and 2. The configuration time specifies the amount of time it takes to configure the HC array for the application execution. These values are quite small and allow the architecture to switch between applications several thousand times per second. Therefore, configuration sequencing and resources reuse becomes practicable with this approach.

The AES implementation delivers the most impressive results. There, the maximum performance without reconfiguration interruptions is in the range of up to 26 MB/s which is very high considering the clock speed of only 100 MHz. Figure 5 shows the running application represented by the HCViewer debugging tool.

The maximum power consumption for the AES256 application, which is the maximum for this application set, is about 150 mW. This value, which was obtained with PrimeTime, includes the dynamic as well as the static power consumptions of the core cells and is right until now an estimated value by the synthesis and layout tools. The exact value will be evaluated once the prototype is finished.

Figure 20 shows the breakdown of the standard cell types for each HoneyComb cell type of the ASIC prototype. It is noticeable that the MEMHCs and IOHCs are mostly composed of integrated SRAM blocks. In case of the MEMHC these memory blocks are the main part of the configurable cell functions. In case of the IOHC the memory blocks are part of the FIFOs for clock domain crossing. The DPHC cells are mainly dominated by the combinational logic required for arithmetic units, multiplexers, and decoders. However, 8% of the design is composed of buffer cells, which is a quite good value considering the fact that we did not optimize design to minimize the use of buffers and did the layout analysis quite conservative.

Figure 21 reflects the area distribution of the HoneyComb architecture. Since the prototype includes 11 DPHCs 57% of the area is allocated by those cells. However, the breakdown of the functional unit (FU) and routing unit (RU) of the DPHCs shows optimization potential for the future. Right now, we used simple multiplex structures to design the architecture; especially the RUs are using multiplex structures extensively. By substituting the multiplexers with more efficient crossbar structures, according to our experiments, we expect to save up to 50% of the area.

9. Conclusion

This contribution presented an application-tailored methodology for a reconfigurable architecture, the HoneyComb

Table 1: Performance results for selected applications at 100 Mhz.

Application	DPHCs	MEMHCs	Config. time	Performance
AES256	11	2	6,85 μs	25,6 MB/s
iMDCT 1xfinger	3	1	24,06 μs	47,6 blocks/s
iMDCT 7xfingers	11	2	25,60 μs	333,46 blocks/s
FFT1024	4	1	7,65 μs	10850 blocks/s
Wavelet	6	1	3,15 μs	0,6 cycles/pixel

Table 2: Synthesis and power evaluation results at 100 MHz.

Application	DPHC area (μm^2)	MEMHC area (μm^2)	IOHC area (μm^2)	Leakage power (mW)	Dynamic power (mW)
AES256	362636	1226638	623680	5.59	146.73
iMDCT 1xfinger	461697	1290972	812529	7.11	66.23
FFT1024	472477	948950	787658	7.26	75.02
Wavelet	250042	1246197	728551	4.2	87.84
ASIC	652285	1299802	868313	10.76	

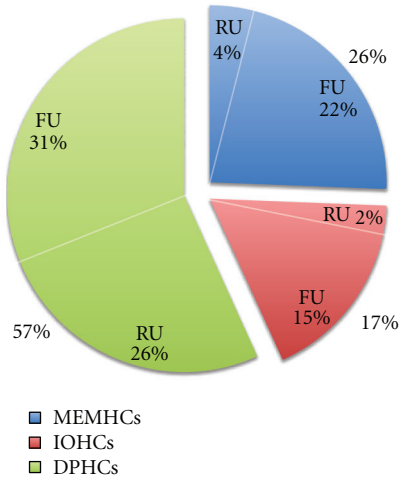


Figure 21: Area distribution across cells within the HoneyComb array prototype.

architecture. Since a fully flexible approach is simply too expensive and usually not desired it is possible to reduce the architecture according to a predefined set of applications. This approach saves silicon area and therefore money. In case of our prototype we reduced the initially required area to less than 50%. The prototype of the HoneyComb architecture has been produced and is already delivered. The layout of the PCB is complete, and the PCB is currently in production. We are expecting to have a running system within the next few months.

The performance results are also promising. Compared to Intel Core 2 Quad processor with 2666 MHz, which reaches about 200 MB/s executing the AES256 application resulting in an overall performance per core of about 50 MB/s, our results are excellent.

Still a lot of work has to be done regarding the area efficiency and programming interface. Currently the used simple multiplexing structures within the array can be replaced by more efficient cross-connect structures. Also, support for at least C is required and would grant access to a wide range of applications.

Acknowledgments

The authors acknowledge support by Deutsche Forschungsgemeinschaft and Open Access Publishing Fund of Karlsruhe Institute of Technology.

References

[1] Xilinx Inc., http://www.xilinx.com/.

[2] Altera Corp, http://www.altera.com/.

[3] J. Becker, T. Pionteck, and M. Glesner, "DReAM: a dynamically reconfigurable architecture for future mobile communication applications," in *Proceedings of the 10th International Conference on Field Programmable Logic and Applications*, Villach, Austria, 2000.

[4] R. Kress, *A fast reconfigurable ALU for Xputers [Ph.D. dissertation]*, Kaiserslautern University, 1996.

[5] T. Oppold, T. Schweizer, J. F. Oliveira, S. Eisenhardt, and W. Rosenstiel, "CRC—concepts and evaluation of processor-like reconfigurable archtitectures," *IT-Information Technology*, vol. 49, no. 3, p. 147, 2007.

[6] A. Abnous, H. Zhang, M. Wan, G. Varghese, V. Prabhu, and J. Rabaey, "The Pleiades Architecture," in *The Application of Programmable DSPs in Mobile Communications*, John Wiley & Sons, Chichester, UK, 2002.

[7] P. Master, "The next big leap in reconfigurable systems," in *IEEE International Conference on Field-Programmable Technology (FPT '02)*, pp. 17–22, December 2002.

[8] E. Schüler and M. Weinhardt, "XPP-III: the XPP-III reconfigurable processor core," *Lecture Notes in Electrical Engineering*, vol. 40, pp. 63–76, 2009.

[9] N. Suzuki, S. Kurotaki, M. Suzuki et al., "Implementing and evaluating stream applications on the dynamically reconfigurable processor," in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '04)*, pp. 328–329, April 2004.

[10] P. M. Heysters, G. J. M. Smit, and E. Molenkamp, "Energy-efficiency of the MONTIUM reconfigurable tile processor," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '04)*, pp. 38–44, Las Vegas, Nev, USA, June 2004.

[11] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Matt, and R. R. Taylor, "PipeRench: a reconfigurable architecture and compiler," *Computer*, vol. 33, no. 4, pp. 70–77, 2000.

[12] D. C. Chen and J. M. Rabaey, "A reconfigurable multiprocessor IC for rapid prototyping of algorithmic-specific high-speed DSP data paths," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 12, pp. 1895–1904, 1992.

[13] C. Ebeling, C. Fisher, G. Xing, M. Shen, and H. Liu, "Implementing an OFDM receiver on the RaPiD reconfigurable architecture," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1436–1448, 2004.

[14] G. Lu, H. Singh, M.-H. Lee et al., "The MorphoSys dynamically reconfigurable system-on-chip," in *Proceedings of the 1st NASA/DoD Workshop on Evolvable Hardware*, pp. 152–160, 1999.

[15] E. Mirsky and A. DeHon, "MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 157–166, April 1996.

[16] T. Miyamori and U. Olukotun, "A quantitative analysis of reconfigurable coprocessors for multimedia applications," in *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 2–11, April 1998.

[17] A. Thomas and J. Becker, "New adaptive multi-grained hardware architecture for processing of dynamic function patterns (Neue adaptive multi-granulare Hardwarearchitektur)," *IT-Information Technology*, vol. 49, no. 3, p. 165, 2007.

[18] A. Thomas and J. Becker, "Multi-grained reconfigurable hardware architecture with online-adaptive routing techniques," in *Proceedings of the IFIP International Conference on Very Large Scale Integration (IFIP VLSI-SOC '05)*, Perth, Western Australia, October 2005.

[19] P. Briggs, *Register Allocation via Graph Coloring*, Rice University, Dissertation, 1992.

[20] C. J. Tseng and D. P. Siewiorek, "Automated synthesis of data paths in digital systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 5, no. 3, pp. 379–395, 1986.

[21] VMEbus International Trade Association (VITA)—FMC Marketing Alliance, http://www.vita.com/fmc.html.

[22] E. O. Brigham and R. E. Morrow, "The fast Fourier transform," *IEEE Spectrum*, vol. 4, no. 12, pp. 63–70, 1967.

[23] St. Mallat, *Phane: A Wavelet Tour of Signal Processing*, Academic Press, 2009.

[24] V. Nikolajevic and G. Fettweis, "New recursive algorithms for the unified forward and inverse MDCT/MDST," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 34, no. 3, pp. 203–208, 2003.

[25] J. Daemen and V. Rijmen, "AES proposal: Rijndael," 1999.

[26] W. J. Cooley and W. J. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, pp. 297–301, 1965.

[27] National Institute of Standards and Technology, http://www.nist.gov/index.html.