QUKU: A Dual-Layer Reconfigurable Architecture

NEIL W. BERGMANN, University of Queensland
SUNIL K. SHUKLA, IBM TJ Watson Research Center
JÜRGEN BECKER, Karlsruhe Institute of Technology

A new architecture, QUKU, is proposed for implementing stream-based algorithms on FPGAs, which combines the advantages of FPGA and Coarse Grain Reconfigurable Arrays (CGRAs). QUKU consists of a dynamically reconfigurable, coarse-grain Processing Element (PE) array with an associated softcore processor providing system support. At a coarse-grain, the PE array can be reconfigured on a cycle-by-cycle basis to change the PE functionality similarly to that in a conventional CGRA. At a fine-grain, the whole FPGA can be reconfigured statically to implement a completely different PE array that serves the target application in a better way. Advantages of the fine-grain reconfiguration include individually customized PEs, adaptable numeric format support and customizable interconnect network. A prototype CAD tool framework is also developed which facilitates programming the QUKU architecture. An example application consisting of two different image detectors is implemented to demonstrate the advantages of QUKU. QUKU provides up to 140 times speedup and 40 times improvement in area-time product compared to an implementation running on an FPGA-based softcore. The area-time product for QUKU is around 16% lower than that of a custom circuit based implementation on the same FPGA. The per-PE customization provides an area-time saving of approximately 31% compared to a homogeneous 4 × 4 array of PEs for the same application. The experimental results demonstrate that a dual layered reconfigurable architecture provides significant potential benefits in terms of flexibility, area and processing efficiency over existing reconfigurable computing architectures for DSP.

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Signal processing systems*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Parallel processors*; B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Algorithms implemented in hardware*

General Terms: Algorithms, Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Field programmable gate arrays, FPGAs, coarse grain reconfigurable arrays, CGRAs, multimedia computing

ACM Reference Format:
Bergmann, N. W., Shukla, S. K., and Becker, J. 2013. QUKU: A dual-layer reconfigurable architecture. ACM Trans. Embedd. Comput. Syst. 12, 1s, Article 63 (March 2013), 26 pages.
DOI: http://dx.doi.org/10.1145/2435227.2435259

1. INTRODUCTION

Many modern, high-performance, multimedia computing applications often require more computational power than is available on a single processor. Many different computing architectures have been proposed for such systems, each with their own

This research was supported by the Australian Government's International Science Linkages Program.
Authors' addresses: N. W. Bergmann, School of ITEE, University of Queensland, st Lucia, Brisbane 4072, Australia; email: n.bergmann@itee.uq.edu.au; S. K. Shukla, IBM TJ Watson Research Center, Hawthorne, NY 10532; J. Becker, ITIV, Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany.

DOI: http://dx.doi.org/10.1145/2435227.2435259

particular advantages and disadvantages. For a specific application, a custom Application Specific Integrated Circuit (ASIC) can provide the optimal area, speed and power performance. However, the expensive Non-Recurrent Engineering (NRE) costs and the long design and verification times associated with state-of-the-art ASIC design often rule them out as an implementation option.

Instead, architectures based on more general purpose computing platforms change the design problem into a programming problem—either software programming in the case of multi-processor platforms, or hardware reconfiguration in the case of Field-Programmable Gate Arrays (FPGAs) and Coarse-Grain Reconfigurable Arrays (CGRAs). This removes the large manufacturing NRE associated with ASICs (or at least shares it amongst all the platform users), and allows more rapid design and verification, so improving time-to-market. Additionally, with programmable solutions, continuing design improvements can be made after the system has been deployed.

Reconfigurable computing architectures, such as FPGAs and CGRAs, are widely promoted as effective implementation platforms for digital signal processing (DSP) and other stream-based applications. However, both these have problems. Runtime reconfiguration in FPGAs is too slow for many applications (order of milliseconds), and implementing dynamic reconfiguration is technically complex. CGRAs have fast reconfiguration (nanoseconds), but the processing elements are often too application specific to be efficient and useful for a wide range of applications. This article particularly investigates a new design technique for implementation of stream-based applications on FPGAs, using architectural templates and programming techniques more normally associated with CGRAs. In effect, a CGRA-like processing-element (PE) array is implemented on an FPGA, which captures the CGRA-benefits of simplified configuration, and fast reprogrammability. Unlike conventional CGRAs, however, the PEs can be customized for each individual application by reprogramming the FPGA at bit-stream level. In this way, it is possible to capture some of the benefits of both technologies.

Because the system is configurable at both FPGA (fine-grain) and CGRA (coarse-grain) level, we have called it a dual-layer reconfigurable array, or a mixed-grain reconfigurable array. The work was conducted as a joint project of University of Queensland, Australia (QU) and Karlsruhe University (now Karlsruhe Institute of Technology), Germany (KU), hence its name QUKU (“cuckoo”).

It is not the purpose of this article to undertake a detailed comparison of the relative advantages of reconfigurable computing architectures versus purely software programmable processor-based platforms, such as multi-core and future many-core architectures, and General Purpose Graphics Processing Units (GPGPUs). Such comparisons are certainly interesting and necessary, and are presented many times in the literature already, with the predictable conclusion that all the different architectural platforms have their own advantages and disadvantages. Instead, this article investigates a new methodology for implementing stream-based processing algorithms on FPGAs. It is taken as given that FPGAs are an effective and widely used implementation style for DSP applications. This work aims to investigate the relative advantages and disadvantages of different ways of implementing stream-based applications on FPGAs.

The article is organized as follows. In Section 2, we review existing stream-based processing implementation styles on FPGAs and CGRAs. In Section 3, we introduce our new QUKU architecture. In Section 4, we describe the implementation of a specific application on QUKU, an edge detector image processing application, and compare the results with other implementation styles. Section 5 analyses the results of these experimental implementations, and gives some overall conclusions.

2. RECONFIGURABLE COMPUTING ARCHITECTURES FOR STREAM-BASED PROCESSING

2.1. Stream-Based Processing

Stream-based data processing is used in a wide variety of applications where data from a continuous audio, image, video, sensor data, radio or communications stream needs to be processed in real-time. The same operations are repeatedly applied to streams of data. A stream can be a simple sequence of digital samples, such as in an audio signal, or it may be a structured two-dimensional, colour-vector sequence of samples representing an image, or a stream of complex objects representing a video stream. These streams of data may need to be processed in real-time, and typically there are many arithmetic operations associated with each sample in the stream.

Stream-based processing provides challenges in both computation and communication for processing architectures. In communications, samples can arrive continuously at high speed, and also partially completed computations need to be stored into and retrieved from local memories and registers. Architectures typically require efficient Input/Output mechanisms, and also require the availability of high overall memory bandwidth. On the computation side, the large number of arithmetic operations often requires some level of parallel processing. Fortunately, the computational nature of stream-based algorithms is generally very amenable to parallel processing. This can be done either by having multiple processing elements associated with the processing for each sample, or by having different processing elements dealing with different samples. In many cases, these two approaches are combined by implementing a processing pipeline, where the processing for one sample is broken into multiple stages, and when a stage has completed its processing, the sample moves to the next stage, and the current stage starts work on the next sample.

FPGAs are well suited to stream-based parallel processing [Woods et al. 2008; Tessier and Burleson 2001]. The necessary arithmetic operators can be constructed from look-up-table (LUT) based logic elements, or from specialized arithmetic processing blocks included in many modern FPGAs. Since every LUT typically has an associated synchronous register, FPGAs are also well suited for building processing pipelines. FPGAs often include small local memories throughout the fabric which allows local operand storage, they have hundreds of I/O pins to allow the connection of many external memories, if needed, for operand storage, and they have a wide variety of input/output ports which can be directly interfaced to the processing pipelines.

Typically, design of an FPGA processing pipeline requires the design of a custom logic pipeline, using a Hardware Description Language such as VHDL or Verilog. Because of the difficulty and designer-time required for custom logic design, many CAD tools to improve designer productivity have been investigated. “System Generator for DSP” from Xilinx is a typical vendor tool [Hwang et al. 2001]. Here, the DSP algorithm is coded using popular modeling tools such as MATLAB and Simulink, with a specialized blockset. Once the processing pipeline has been correctly modeled and debugged in the software pipeline, it can be automatically mapped to specific Xilinx FPGA implementations of the blockset.

This approach works well if the entire pipeline fits onto the FPGA, and if the pipeline processing is uniform. The block-based approach is much harder to use, if the required functions change during processing. Modern FPGAs have the ability to be dynamically reconfigured during execution, so it is possible to change the functionality of the pipeline during runtime [Lysaght et al. 2006]. However, despite several decades of research, dynamic reconfiguration remains technically complex and often computationally inefficient due to long, serial configuration bitstreams. To date, stream-based design tools such as “System Generator for DSP” do not incorporate any support for dynamic reconfiguration.

2.2. Coarse Grain Reconfigurable Arrays

Coarse-grain reconfigurable arrays (CGRAs) consist of an interconnected matrix of multiple word-level function units [Todman et al. 2005]. These function units, commonly referred to as processing elements (PEs), may be as simple as an ALU or may be a highly complex specialized unit capable of executing a complete algorithm. The PEs are interconnected using a programmable interconnect network. The function of the PEs and the interconnect network are controlled by configuration data. Configuring system operations at this coarse grain level significantly reduces the size of the configuration code compared to FPGAs. The short configuration codes enable very fast dynamic reconfiguration. Generally, the PEs in CGRAs have been designed for a specific application-domain. Within that particular application domain they are efficient but they may not be flexible enough to adapt to other application domains. The amount to which they can be extended to other application domains determines their flexibility. A number of classic and contemporary CGRA architectures are described in the following sections.

2.2.1. CGRA Architectures. MorphoSys is a coarse-grain, integrated reconfigurable system-on-chip targeted at high-throughput and data-parallel applications [Lee 2000; Singh 2000]. The targeted applications are multimedia and image processing. MorphoSys is a reconfigurable computer architecture that is composed of a processing unit called TinyRISC, a reconfigurable cell array (RC Array), context memory, frame buffer and a DMA controller. Each RC consists of an ALU/Multiplier and other control logic. The Tiny RISC processor core has an extended instruction set for effectively controlling the MorphoSys RC Array operations. These instructions enable data transfer between main memory (SDRAM) and frame buffer, load configuration from main memory into context memory, and control RC array execution. The DMA provides a fast interface between memory and the system. The frame buffer (FB) is a local memory buffer. The RC Array has 64 reconfigurable cells arranged in an 8 by 8 array. Each cell has an ALU/MAC unit and a register file. The RC array functionality and interconnection network are configured through context words. The RC Array has a three-layer interconnection network. The first layer connects the RCs in a two-dimensional mesh, allowing nearest neighbor data interchange. The second layer provides complete row and column connectivity within an array quadrant. It allows each RC to access data from any other RC in its row/column in the same quadrant. The third layer supports inter-quadrant connectivity.

The Garp CGRA was proposed by the BRASS group at the University of California, Berkeley [Hauser and Wawrzynek 1999]. The Garp Architecture combines reconfigurable hardware with a standard MIPS processor on the same die to retain the best features of both. The idea behind Garp is to identify the repetitive computationally intensive structures in the software and map them on the reconfigurable hardware to achieve algorithm speedup. Garp contains a RISC processor core with extended MIPS-II instruction set. The reconfigurable unit consists of entities called blocks. The array is organized as 32 rows by 23 columns of 2-bit logic blocks. A 24th column of control blocks manages communication outside the array. Each logic block takes as many as four 2-bit inputs and produces up to two 2-bit outputs. Each row can be configured to perform any 4-input logical function, a 3-input addition/difference, or a variable shift, on up to 46 bits of data. The control block acts as an interface between the logic blocks and the outside world. It can interrupt the processor and can also initiate transfers to and from the data memory. Each block in the array requires exactly 64 configuration bits to specify the sources of inputs, the function of the block, and any wires driven with outputs. With a 128-bit wide interface to external memory, loading a full 32-row configuration takes 384 sequential memory accesses. Partial reconfiguration of the array is possible in row increments.

Chameleon is an energy efficient heterogeneous System on Chip for mobile computing applications [Heysters 2002]. A Chameleon system consists of heterogeneous processing tiles that are connected to each other by a Network on Chip. The idea is to map different tasks on the 32 tiles best suited for the requirements. The aim is to achieve maximum energy efficiency for mobile applications. The heterogeneous architectural template of Chameleon consists of four different kinds of processing tiles: general purpose (GPP/DSP), fine-grained reconfigurable (FPGA), coarse-grained reconfigurable (DSRA) and application specific (ASIC). A complete architectural template consists of 16 processing tiles interconnected using a network-on-chip. There is a real time scheduler, called the Central Configuration Manager running on one of the GPP, which synchronizes the inter-tile processing of task, and decides on runtime scheduling. There is a library for common tasks, such as FFTs and convolutional coders, which may include multiple implementations for the same task suited for different tiles. One part of this architecture (DSRA) has now been commercialized as the Montium processor from Recore Systems [Recore Systems 2007], however it is not used as a CGRA. Rather one Montium processor is typically used as an application specific DSP accelerator, such as an FFT or Viterbi decoder, within an SoC.

The eXtreme Processing Platform (XPP) from PACT Corporation claims to combine the flexibility of a programmable processor with the speed of an ASIC [Baumgarte et al. 2003]. It is targeted for applications in multimedia, telecommunications, simulation, digital signal processing, cryptography and similar application domains. The XPP platform consists of an array of coarse grain elements called processing array elements (PAEs). The core consists of three types of PAEs—arithmetic elements, memory access elements, and processor elements. The bus backbone contains an N bit data bus and 1 bit event bus. The data bit-width can be chosen between 16, 24 and 32 bits.

Quicksilver's adaptive computing machine (ACM) is a collection of adaptable heterogeneous computing engines, called nodes, which are connected using an adaptable network [Master 2002]. The target domain is low power wireless applications. ACM is based on the concept that applications are heterogeneous in nature and so the architecture should be also. The ACM architecture consists of heterogeneous entities of mixed granularities, called nodes. Each node is self-sufficient with resources such as a controller, memory and functional unit. A node can be complex enough to have the capability of executing an algorithm on its own. The nodes are classified as adaptive, domain or programmable. The adaptive nodes are flexible but domain-specific. The domain nodes are specialized to do a certain task, similar to ASICs. The programmable nodes are flexible but offer less processing power.

The Dynamically Reconfigurable Processor (DRP) is a coarse-grain reconfigurable processor core developed by NEC [Toi et al. 2006]. The primitive unit of the DRP is a tile and each tile consists of 8×8 processing elements, state transition controller, 16 dual port vertical memories and 8 single port horizontal memories. Each PE has an 8 bit ALU, 8 bit Flip-flop and an 8 bit $\times$ 16 word register file. The DRP core is scalable and can consist of multiple tiles. The core can be reconfigured every cycle. DRP-1 was the first prototype developed by NEC and consists of an 8 tile DRP core, eight 32-bit multipliers, an external SRAM controller, PCI interface and 256-bit I/Os. The operating frequency was 133 MHz.

A new alternative to CGRAs is the general purpose, or application-specific many-core architecture: effectively a large number of fairly conventional CPUs on a single die. For example, the aSAP platform [Baas et al. 2007] contains a 6×6 array of 16 bit CPUs optimized for DSP applications. Such architectures are a relatively poor fit to FPGAs. Existing FPGA softcore CPUs such as Xilinx's Microblaze processor are relatively large and slow, and the comparison of the relative processing efficiencies of Microblaze and QUKU later in the article suggests a simplistic many-core architecture would be inefficient for FPGA-based DSP.

From the above architectural survey, some key observations can be made to help inform our design of QUKU. Firstly, architectures tend to consist of three main components: a core array of PE tiles, a coordinating RISC processor or similar to deal with the interface of the CGRA to other subsystems, and specialized memory interface units. Therefore our first QUKU prototype will adopt this general architecture.

Secondly, there is wide variation in word-size. Some architectures have a fixed small granularity (e.g., 8 bits for DRP, 2 bits for Garp) which can be combined for larger words; others allow an array to be built with different word sizes (e.g., XPP, Montium). This indicates that an array needs some mechanism to cope with different word sizes. Unlike existing CGRAs, a dual-layer reconfigurable system such as QUKU allows the word-size to be customized on a per-application basis.

Thirdly, architectures vary in whether a homogeneous or heterogeneous array of tiles is used. However, in all existing CGRAs this needs to be decided at fabrication time. [Van Essen et al. 2011] reviewed the advantages of specialized functional units versus homogeneous functional units in the Mosaic architecture and found that carefully chosen specialized units could provide approximately a 15% advantage in area-delay-energy product for benchmark applications. Application-specific specialization of PEs is easily achieved in a dual-layer reconfigurable architecture such as QUKU, and so this will be explored to see if similar savings can be achieved.

2.2.2. CGRA Programming. In the same way that an architectural survey of CGRAs can assist in the architectural design of QUKU, so a survey of CGRA programming methods can assist with the design of the toolchain for QUKU.

The MorphoSys design flow is supported through a GUI called mView [Lee 2000] that takes user input for each application. The specifications include data source, destination, and operations. mView generates assembly code for the RC Array. mView has several built-in features that allow visualization of RC execution, interconnect usage patterns for different applications, and single-step simulation runs with backward, forward and continuous execution.

The Berkley group has a retargetable compiler for Garp [Callahan 2002]. The reconfigurable array configuration is coded in a file in a simple textual language. This source is fed through a program called the configurator to generate a representation of the configuration as a collection of bits. A simulator is available for program development.

The design methodology for the Chameleon SoC is based on the Kahn Process Network model [Smit et al. 2004]. The kernels in the applications are replaced by remote procedure calls (RPCs). Then the processes are implemented using one or more tiles. Compiling processes in C for GPPs is straightforward but additional efforts are required for translation of processes from C to executable code for specialized processors. Coding the Montium core is sufficiently difficult that pre-packaged, pre-programmed cores are the most successful commercial product.

The XPP design flow starts with profiling of a C/C++ code on a processor element [Baumgarte et al. 2003]. Depending upon the profile parameters the application is divided into parallel threads to exploit the parallelization in application and the hardware.

The INSPIRE software development kit is used for developing applications to be implemented on Quicksilver's ACM [Master 2002]. INSPIRE comprises the SilverC high-level design language, simulator, compilers, assemblers, debuggers, and utilities for creating end-to-end solutions for ACM-based platforms. The design flow enables developers to express system functionality as software, without considering hardware partitioning, task threading, or memory allocation.

DRP comes with an integrated design environment that includes a high level synthesis tool, a design mapper, simulator and a layout/viewer tool [Toi et al. 2006]. Applications can be written using a high level C language description that is synthesized and mapped on the DRP processor by the tools.

In terms of the design flow for QUKU, a full design suite needs ways to capture the designer's intent in a high-level format in terms of the required arithmetic operations—perhaps something like C or MATLAB. Next these need to be converted to a register-transfer level list of individual operations and their data dependencies. In QUKU, a Data-Flow Graph notation will be used. There is then a mapping phase, where operations are mapped in time and space onto the array, and appropriate configuration codes and schedules generated.

Many of these algorithms already exist for other CGRAs. As described in the architecture section above, most CGRAs are not a stand-alone device, but rather are one component in a more complex system, which typically includes a coordinating RISC processor. There is therefore a need for hardware-software partitioning, i.e., identifying those parts of the system where most benefit is gained from moving to a CGRA. The QUKU approach is to use profiling tools [Westcoff and Docef 2007] to identify computation kernels that require speedup. Having decided which algorithms will be implemented on the CGRA, the next step is mapping. [Ahn et al. 2005; Yoon et al. 2008] present detailed analyses of sophisticated spatial and temporal mapping tools for CGRAs. It is shown that the mapping problem is NP-complete, but heuristic solutions can provide very good results. In QUKU, our initial mappings will be done manually, but these other research groups have shown that these tasks can be effectively automated.

2.2.3. Discussion. This section has detailed a number of well-known CGRAs—many more proposed architectures are surveyed in Todman et al. [2005] and Oppold et al. [2007]. These CGRAs have shown excellent computing performance and efficiency. At the least for the commercial products, the main market has been as a component on SoCs for efficiently implementing a specific range of DSP functions as part of a larger system. CGRAs have not been successfully developed as a stand-alone chip that system designs can incorporate at board level.

CGRAs gain computational efficiency by making some particular system choices at design time. The granularity of the CGRA tiles needs to be fixed. The supported arithmetic operations (e.g., fixed point or floating point) also need to be fixed, as do the interconnection network, the I/O network and the memory interfaces. Because DSP operations vary so greatly, it is hard to find a particular set of array options that suits a wide-enough set of applications for the approach to be viable as a stand-alone product.

On the positive side, the published experimental and commercial architectures have shown that for specific applications, a CGRA can be an effective application accelerator, usually in combination with a conventional processor. Fast reconfiguration of tiles, often just a few clock cycles, allows rapid re-use of tiles for different stages of an algorithm. Building a DSP accelerator becomes a programming exercise, which is much simpler than a hardware design. CAD tools allow rapid design-space exploration, and convert programs, often naturally expressed as some sort of data flow diagrams, or perhaps as C-like programs, into CGRA configurations.

FPGAs are already a popular target for DSP implementation. A typical FPGA system might consist of a processor (either external, on-chip hardcore processor, or on-chip softcore processor), plus an application-specific accelerator, perhaps designed using a tool such as System Generator for DSP. Given the effectiveness of CGRAs as a DSP accelerator for SoCs, we have been motivated to explore the potential of CGRA-like accelerators for stream-based applications on FPGAs. CGRAs compare favourably in terms of speed, area and power efficiency with custom DSP circuits on SoCs, especially where the CGRA can implement multiple functions during different stages of the algorithm. We might expect similar advantages on an FPGA.

Compared to a conventional CGRA on an SoC, using an FPGA to implement a CGRA has several potential advantages that this project also explores. Firstly, the CGRA tiles can be chosen to have an optimal word-size and arithmetic type, and the CGRA array size can also be selected on an application-by-application basis. This can be done dynamically, at fine-grain level, by reprogramming the FPGA between applications, with a time delay in the order of hundreds of milliseconds.

FPGAs also present a second level of customization that is not possible with a fixed CGRA. For a given suite of algorithms, the mapping of operations and communication to CGRA tiles and I/O links will be known at FPGA configuration time. Any unused tiles in the array do not need to be implemented. Any unused features of remaining tiles can be deleted before the FPGA bitstream is generated (e.g., if a tile only does ADD and never MULTIPLY, the multiplier can be deleted). Also any unused communications links can also be automatically deleted before FPGA programming. Part of the work in this project is to explore the area gains from such individual tile-by-tile customization.

Because of the increasing large bitstream sizes in modern FPGAs, schemes for multi-level reconfigurable architectures have appeared in recent years [Lange and Middendorf 2008]. QUKU represents a similar attempt to allow more raid reconfiguration during algorithm execution.

3. QUKU ARCHITECTURE, CONFIGURATION AND OPERATION

This section describes the key features of the QUKU architecture. The general approach that has been taken is to base QUKU on those features that are common to a number of different CGRAs, and have demonstrated their efficiency for stream-based algorithm implementation. The architecture has been customized for FPGAs, in the sense that the architecture needs to take account of the limited availability of resources such as on-chip memories.

In common with many CGRAs, QUKU consists of a regular PE array that is designed to provide stream-based algorithm speedup. Also, like many CGRAs, the QUKU PE array is not designed to be a stand-alone target for complete applications. Rather the PE array is paired with a conventional microprocessor core that handles control-intensive portions of the application, interfaces to file systems, and manages the configuration of the PE array.

3.1. QUKU Architecture

Figure 1, Figure 2 and Figure 3 show the block level description of QUKU. QUKU comprises a processor sub-system, based on a Microblaze softcore, loosely coupled to a dynamically reconfigurable coarse grain PE array. The interface is defined as loosely coupled because the there is no shared memory between the processor and PE array, and communication is done using message passing. An alternative would be a tightly coupled architecture, such as ADRES [Mei et al. 2003]. However, since there are many softcore and hardcore processors for FPGAs, often manufacturer specific, a loosely coupled approach gives more flexibility for our experimental system. Figure 2 shows more detail of the “QUKU PE array” in Figure 1, while Figure 3 expands the internal detail of one PE in the PE array of Figure 2. The individual components in Figure 1, Figure 2, and Figure 3 are each explained in more detail in Sections 3.2, 3.3 and 3.4.

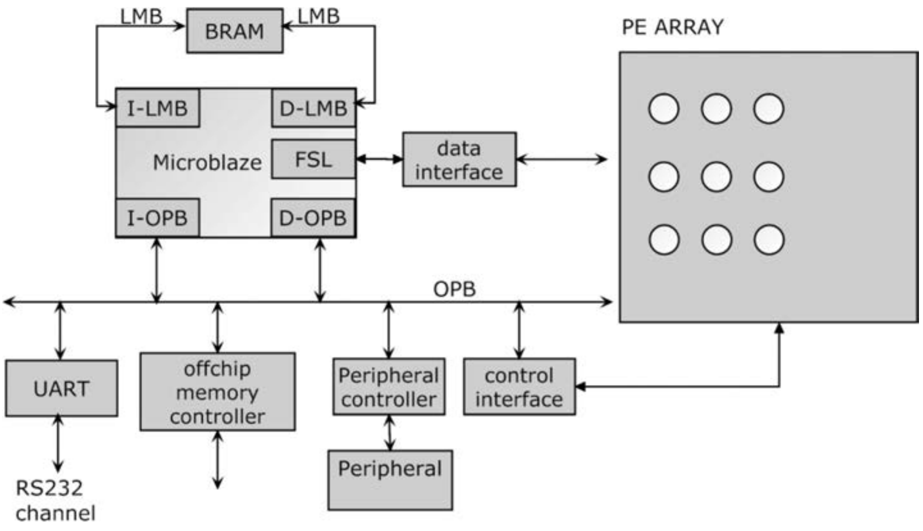


Fig. 1. QUKU block diagram.

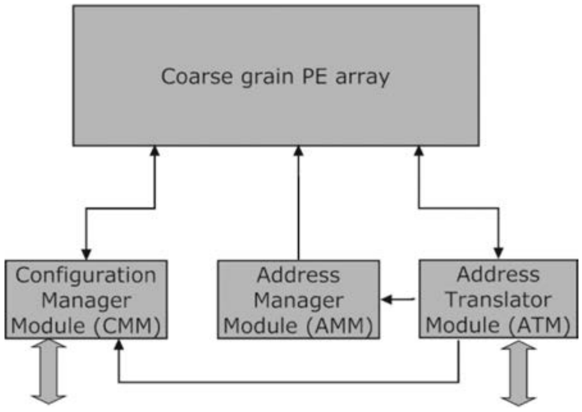

Coarse grain PE array
Configuration Manager Module (CMM)
Address Manager Module (AMM)
Address Translator Module (ATM)

Fig. 2. CGRA block diagram.

3.2. The Processor Sub-system

In QUKU, the processor is responsible for scheduling the computationally intensive part of the algorithm, also referred to as kernels, on the PE array and running the control intensive part of the user application code. A key decision is how to implement the processor: should it be a softcore or a hardcore processor. Although softcore processors are inferior to hardcore processors in terms of performance and power consumption, they offer high flexibility and customizability. An important point in deciding the processor was that the QUKU architecture is not processor-centric. Although the processor efficiency has an impact on the overall processing efficiency, the processor's primary role in QUKU is synchronizing hardware tasks and executing control parts of the application code.

The Xilinx Microblaze soft processor core was selected to act as the central processor [Xilinx 2007]. Microblaze is supported by the Xilinx embedded development kit (EDK) software. The processor's feature set is fully customizable with optional support for single precision floating point arithmetic, which allows the architecture to fit even in a

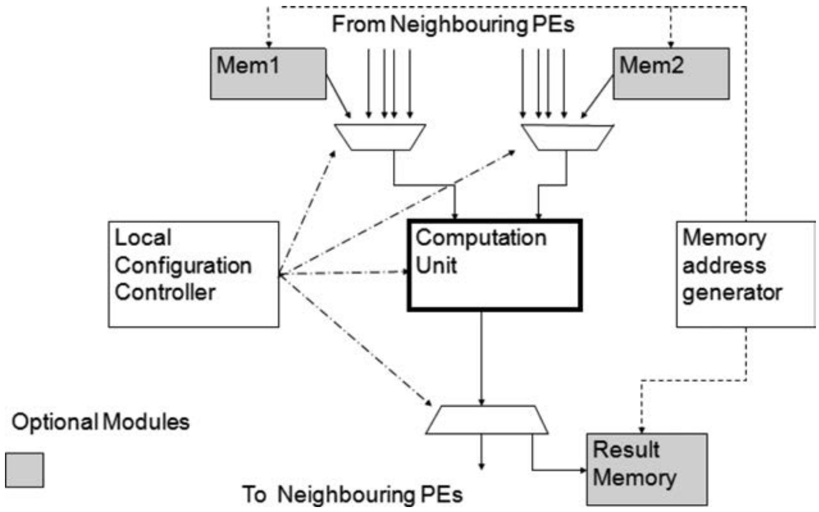


Fig. 3. PE structure.

relatively small FPGA. Microblaze is a 32-bit reduced instruction set computer (RISC) soft-processor core, based on Harvard architecture, optimized for implementation in Xilinx FPGAs.

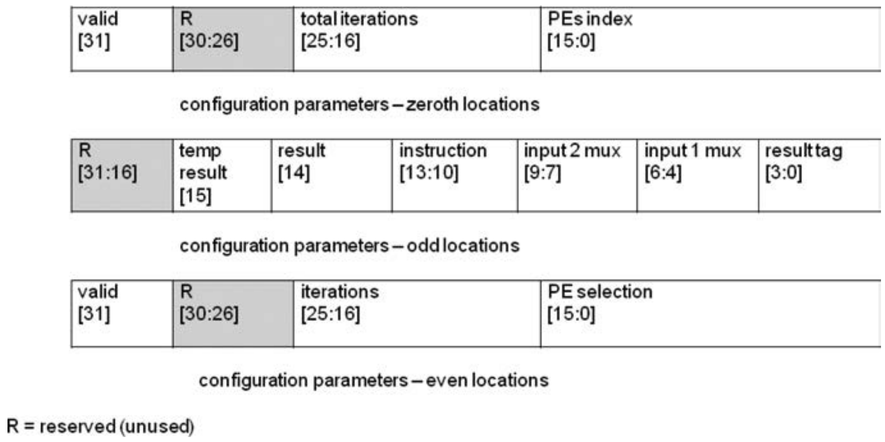
3.3. The QUKU Coarse-Grain Reconfigurable Array

The coarse grain reconfigurable array in QUKU consists of a collection of heterogeneous PEs, the configuration manager module (CMM), the address manager module (AMM) and the address translator module (ATM). The CGRA block diagram is shown in Figure 2. The address translator module in Figure 2 performs the data interface function between the PE array and the host Microblaze processor using a dedicated FIFO link called a Fast Simplex Link [Rosinge 2004] as shown in Figure 1. The Configuration Manager Module performs the control interface between the PE array and the host processor using the standard system bus for the Microblaze system as shown in Figure 1.

3.3.1. Address Translator Module. All the memory blocks inside the CGRA have a fixed memory map following a 32-bit address space. The address translator module (ATM) acts as a bridge between the CGRA and the processor. All the streaming data for the PE array and control information for CMM and AMM are transferred through this module. The processor sends out data and the corresponding address for each read and write transaction. The ATM translates the 32-bit address coming from the processor into module specific address information.

3.3.2. Address Manager Module. The address manager module (AMM) consists of a controller module and a dual port Block RAM (BRAM) to store the address parameters for multiple kernel configurations. Port A of the memory has a write interface and is controlled by the Microblaze processor. Port B of the memory has a read interface and is controlled by the AMM.

3.3.3. Configuration Manager Module. The configuration manager module (CMM) is responsible for storing and loading the PE configurations. It consists of a configuration controller module and a block RAM. The BRAM stores PE configurations for multiple



valid	R	total iterations	PEs index
[31]	[30:26]	[25:16]	[15:0]

configuration parameters – zeroth locations

R	temp result	result	instruction	input 2 mux	input 1 mux	result tag
[31:16]	[15]	[14]	[13:10]	[9:7]	[6:4]	[3:0]

configuration parameters – odd locations

valid	R	iterations	PE selection
[31]	[30:26]	[25:16]	[15:0]

configuration parameters – even locations

R = reserved (unused)

Fig. 4. Configuration memory data structure.

kernels. The controller module maps the configuration onto the PEs on a cycle-by-cycle basis. The configuration memory data structure is shown later in Figure 4.

3.4. The QUKU PE Structure

QUKU consists of a MIMD (multi-instruction multi-data) style PE array where each PE has a mix of processing capabilities that is solely decided by the application requirement. The dimensions of the PE array can be chosen to fit the application requirements, and also the available logic resources on the FPGA. The PE block diagram is shown in Figure 3. Each PE consists of mandatory modules such as the computational unit (CU), local configuration controller (LCC) and memory address generator (MAG) and optional modules such as the input data memory and output result memory.

The PE array is the computational heart of QUKU. A key design decision is the granularity of the computational array. For the low-level FPGA the granularity of the array is given by the technology at bit-level operations. The granularity of the coarse-grain array is part of the QUKU architectural design. For conventional coarse-grain arrays, array elements may be 4 bits, 8 bits, or larger in width. Because the word-length of QUKU can be changed on a per-application basis there is not the same architectural constraint to decide on a fixed bit width which is a compromise between flexibility and efficiency. The most efficient bit-width for PEs is the data word-length, and so QUKU PEs are always chosen to be at word-level granularity, given that this word-length can be changed for each new design.

3.4.1. Memory Address Generator (MAG). Each PE can have up to two input memories, for storing input data, and one result memory for storing the result of the operation. The MAG module stores addressing parameters for the two input data memories and the result memory. Each memory is individually controlled and can be selected or omitted at compile time. Apart from inclusion and exclusion control of memory resources, the memory dimension (width and depth) and implementation (BRAM or distributed memory) can also be controlled at compile time. This application specific memory control allows efficient utilization of finite memory resource in FPGAs. It also reduces unnecessary power consumption.

3.4.2. Local Configuration Controller. Each PE has a local configuration controller (LCC) module to manage the configuration locally. The configuration of a PE can be changed on a cycle-by-cycle basis. The LCC has two register arrays to store the configurations. One array stores the configuration data and the other array stores the iterations associated with each configuration plane. More details about configuration formats are given in the following section.

3.4.3. Computation Unit. The computation unit (CU) is the heart of the PE, as shown in Figure 3. The arithmetic functions that a PE performs are specified at design time. Only those functions that are actually used in any of the different configurations for the given application need to be implemented. This leads to a hardware-optimized implementation.

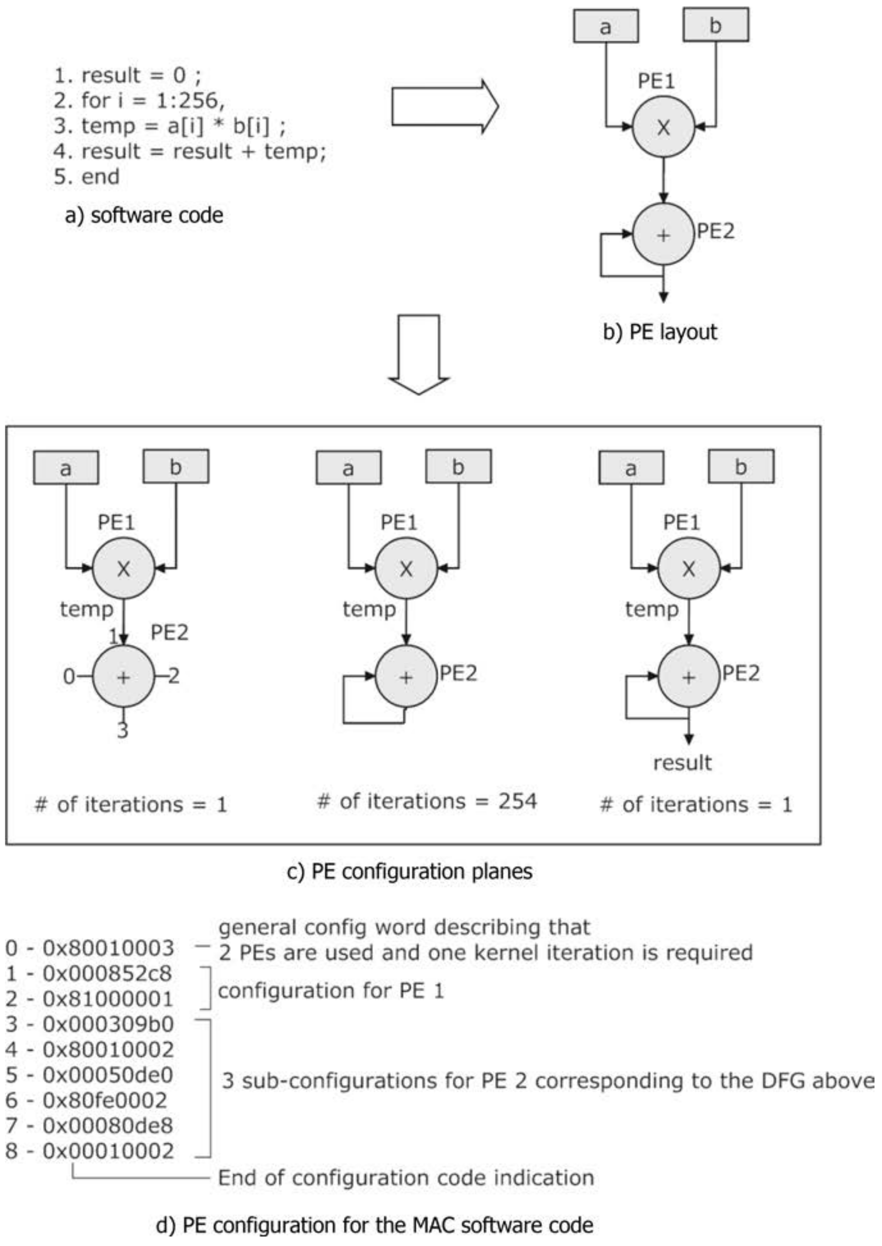
Each PE optionally has two input data memories and a result memory. In most of the algorithms, only the PEs located at the top row of the array receive input data from memories; all the intermediate PEs receive input data from their neighbouring PEs. Similarly, the result memory is mostly used by the PEs located at the bottom row of the data flow graph. Each PE can accept data input from one of the six sources; two input memories and four neighbouring PEs. The right hand side (RHS) input can also be the result of the previous operation. The arithmetic function(s) that a PE executes is specified by its local configuration.

There are many possible design choices for the PE array. Given that our intention is to customize each PE for a particular set of applications by removing unused features, the node design starts with a fairly full-featured node in terms of interconnections to neighbouring nodes. We also have designed the architecture so that it is relatively easy to change features like word length or arithmetic type.

Figure 4 shows the configuration memory structure. The zeroth location in a configuration region contains general information. Bits [15:0] contain information about all the PEs taking part in that configuration. Bits [25:16] contain the number of iterations for which the configuration is valid. Bit 31 at all the even addressed locations indicate the valid information. The first even numbered location where the bit 31 is set to '0' indicates end of configuration. Location 1 and all the subsequent odd addressed locations contain information about the PE(s) configuration. Location 2 and all the subsequent even addressed locations contain information about the PEs for which the configuration is valid. This information is coded in bits [15:0]. Bits [25:16] contain information about the number of iterations for which the configuration has to execute. Bit 31 indicates the end of configuration code. Reserved fields marked "R" in Figure 4 are not currently used.

Every clock cycle, a PE gets a new configuration code from the LCC. One design decision is the maximum allowable configuration code size. For the examples presented in the article, and also for other DSP applications such as FIR filter and FFT, the maximum code size fits within 64 locations, and so this size was chosen for the prototype implementation. Figure 5 illustrates the compaction of configuration code using the example of a simple multiply-accumulate (MAC) core used by many DSP algorithms. Figure 5(a) shows the MAC code over a block of 256 data samples. The corresponding DFG and the resultant PE structure is shown in Figure 5(b). Figure 5(c) shows the execution cycle over 256 iterations.

There are two arithmetic operations in a MAC block: multiplication and addition. The multiplier block iterates over all data items and produces the result. But the adder undergoes three different configurations for the entire iteration limit. For the first iteration, the adder accepts data from input 1, passes it through and stores it into the internal result FIFO. Over the next 254 iterations, the adder adds the incoming data at input 1 with the intermediate result from the result FIFO and updates the intermediate result. For the last iteration, the adder adds the data at input 1 with the intermediate result and updates the final result in the result FIFO and tags it for use by neighbouring PEs or writes it into the result memory. Figure (d) shows



```
1. result = 0 ;
2. for i = 1:256,
3. temp = a[i] * b[i] ;
4. result = result + temp;
5. end
```

```
0 - 0x80010003  — general config word describing that
                  2 PEs are used and one kernel iteration is required
1 - 0x000852c8  ] configuration for PE 1
2 - 0x81000001  ]
3 - 0x000309b0
4 - 0x80010002
5 - 0x00050de0    3 sub-configurations for PE 2 corresponding to the DFG above
6 - 0x80fe0002
7 - 0x00080de8
8 - 0x00010002
    └────────── End of configuration code indication
```

Fig. 5. Multiply-accumulate example software code, the data flow graph and the configuration code.

the configuration code corresponding to the DFG described in Figure 5(e) as follows: #0 (Instruction 0) defines the number of iterations for the whole configuration, #1 specifies the data sources, sink and instruction (multiply) for PE1, #2 specifies that the previous instruction relates to PE1 for all iterations, #3 specifies the first configuration for PE2 that initializes the accumulator, #4 identifies that this applies to PE2 for 1 cycle, #5 and #6 specify the operations for the central configuration of PE2 for 254 cycles, #7 and #8 specify one iteration of a new configuration of PE2 to output the answer, and a "valid" bit of 0 in #8 also marks the end of the configuration.

3.5. Dual Layered Reconfiguration

QUKU combines the architectural concepts of CGRA and FPGA. This dual layered reconfiguration differentiates QUKU from both FPGAs and CGRAs. FPGAs, once reconfigured, retain the circuit till reconfigured again. Reconfiguration in FPGAs can implement a completely different circuit. In CGRAs, the PE array can be reconfigured on instruction-by-instruction basis but this reconfiguration can change the behavior only within the bounds of PE capabilities. QUKU offers the combined benefits of FPGAs and CGRAs.

3.5.1. FPGA Level Reconfiguration. FPGA level reconfiguration allows an application to use a PE that is customized to best fit the range of individual algorithms that are part of that application. FPGA level reconfiguration can also be used to load different PE arrays to support different numerical formats. For example, applications that require greater computational accuracy and flexibility can benefit from floating point representation whereas for applications where the dynamic range of data is known and limited, fixed point arithmetic will often be sufficient. In QUKU, a library of PEs has been developed consisting of fixed and floating point PEs. The FPGA can be statically reconfigured at the cost of a few hundred milliseconds to replace the existing PE array with fixed point capability with that of a floating point capable PE array or vice-versa. Static reconfiguration can also be used to adjust the data width to suit the application requirements. The slow fine grain reconfiguration, which occurs only at application start up, allows a better throughput and processing power and a decrease in power consumption during application execution compared to a fixed PE array.

3.5.2. CGRA Reconfiguration. QUKU consists of an array of PEs whose function can be changed every cycle by CGRA-level reconfiguration. This reconfiguration is used to determine the specific operations of individual PEs during the execution of a particular algorithm (as shown in Figure 5), and it can also be used to change the whole algorithm being implemented, as will be shown in the example application later.

3.6. Design Methodology

One of the key requirements of a new computing architecture is the ability to program that architecture quickly and efficiently for a given application. Although the aim of this project is not to deliver a mature CAD tool suite, a viable design pathway does need to be proposed. QUKU offers a fairly standard approach towards hardware-software partitioning of the user application and programming of the coarse grained array. The computationally intensive part of the user application is targeted for execution on the PE array while the control intensive part is executed on the processor. This section describes the partitioning process, methods of programming the PE array and software-hardware synchronization. Figure 6 gives an overview of the design methodology used in QUKU.

3.6.1. Application Profiling and Partitioning. The first step towards implementation is to write a high-level description of the application. This code is then profiled to find the key areas where most of the execution time is spent. Application code profiling at the source level identifies the potential areas for optimization and speed-up. Westcoff and Docef [2007] present a list of tools and techniques for code profiling and optimization.

In QUKU, a MATLAB-based approach is adopted to generate the profiling results and performance benchmarking. The application is written as a MATLAB script. The profiler tool in MATLAB is used to generate the profiling results. Profiling needs to be

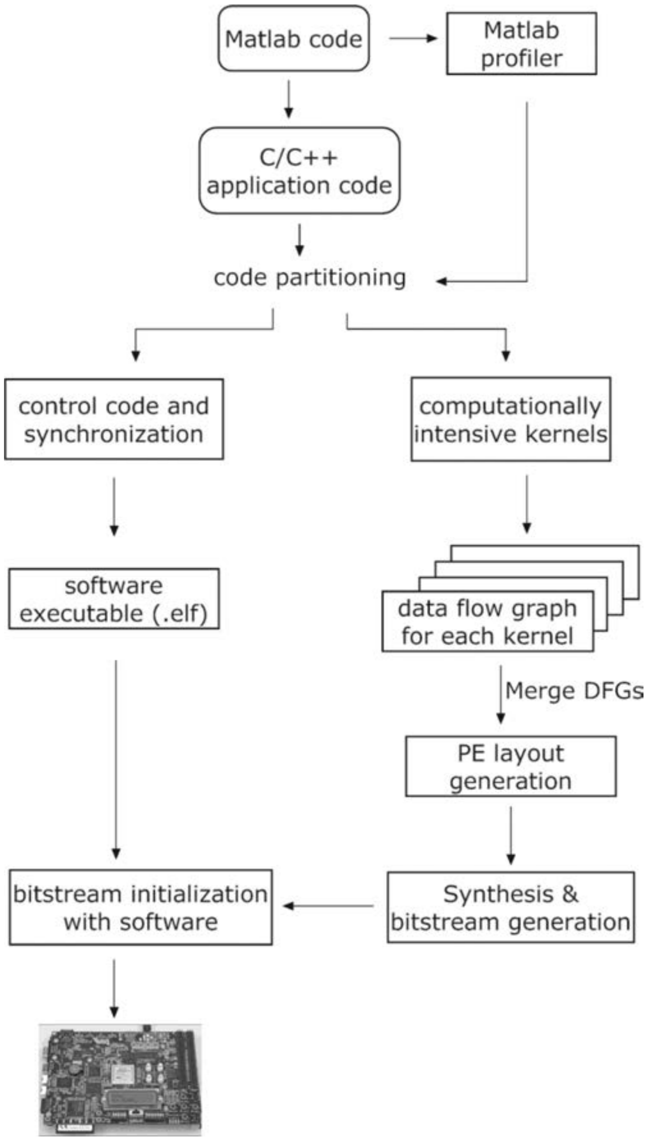

Matlab code
Matlab profiler
C/C++ application code
code partitioning
control code and synchronization
computationally intensive kernels
software executable (.elf)
data flow graph for each kernel
Merge DFGs
PE layout generation
bitstream initialization with software
Synthesis & bitstream generation

Fig. 6. Software-hardware design flow in QUKU.

done at the level of individual arithmetic operations and groups of operations organized into functions. MATLAB proved very useful for this level of profiling. With the profiling results and a manual inspection of the source code, the source code can be partitioned into control oriented, irregular parts and computationally intensive parts. The control oriented and the irregular parts of the code are marked for execution on the Microblaze processor while the computationally intensive parts of the code, referred to as kernels, are marked for execution on the PE array.

After code partitioning, a two pronged, software-hardware approach is taken. On the software side, the user application is modified to remove the kernel code and include synchronization information. On the hardware side, a visual programming approach using data flow graphs is used to generate the hardware (i.e., the CGRA configurations). The software and hardware flow is described in the following sections.

3.6.2. Software Flow. A C/C++ implementation of the completed algorithm is first used to verify that the application gives correct results. This can also be used later to benchmark application speedup. This C/C++ software code is then modified based on the profiling results. The computationally intensive part of the software code is removed and anchor points are inserted. The anchor points are hardware specific information for managing data transfer between the processor and the PE array and achieving synchronization between the software and hardware.

Along with the synchronization information, PE array configuration is also done through software running on the processor. In most cases, the PE array programming is a one-time process that happens in the start up phase. Generally, the first few lines of software code contain the instructions to copy the configuration code for all the kernels, stored in the Microblaze instruction memory, into the PE configuration memory. Switching between configurations at run time is managed by the hardware configuration controller without further software intervention.

3.6.3. Hardware Flow. The hardware flow describes the data flow graph generation and the steps to generate an application specific PE layout. It also includes the configuration code generation of each kernel from the corresponding data flow graph. The starting point for hardware flow is the code that needs to be mapped on the PE array for faster execution.

DFG generation. A data flow graph (DFG) is a very convenient way of describing the flow of streaming data in a signal processing system. It also describes the inter-process dependencies at macro level and the inter-operation dependencies at micro level. In QUKU, a DFG is prepared for all the kernels to be mapped on the coarse grained PE array. In conventional coarse grain architectures, the DFG is mapped on the processing array, taking account of the available features on the PEs. QUKU takes an opposite approach. The DFGs are prepared first assuming a PE with all available functions, and the final PE array layout is generated. The advantage of this approach is that the mapping algorithm has maximum flexibility for operator placement, since all nodes are homogeneous. The disadvantage is that the final size of the PE array hardware (once all unused features are removed) is not known at the start. Therefore, there may be a need to iterate the design process with different array sizes to find the best array that fits on the FPGA and gives best speed performance. This approach has proved to be area, speed and power efficient in the example shown later.

Merging of DFG. Individual DFGs will be generated for each different kernel within a single application. Once all the individual DFGs are prepared, they are combined to give a final DFG that includes all of the functions for all the kernels. The total DFG now needs to be mapped to physical PEs in time and space, while at the same time deciding what combination of physical PEs are available for mapping. This is a classical Design Space Exploration (DSE) problem, and optimization of DSE is an active research area in the CAD community. In terms of the domain space for the mapping problem, it is similar to the problem size for a heterogeneous CGRA since operations are mapped to PEs in time and space, accounting for the capabilities of each node. For our simple examples, this was done manually, and could be done quite quickly without multiple iterations. Since the merged graph for this example was the same size as the largest of the individual graphs, the solution is a good one. As the size of the graphs grows, more detailed analysis of the merging algorithm would be needed to evaluate its performance and optimality. Efficient heterogeneous CGRA mapping algorithms have been demonstrated by other research groups [Yoon et al. 2008].

Typically, the DSE optimization problem for QUKU is to minimize the execution time of all the DFGs, subject to the constraint that the PE array has to fit on the available FPGA resources. In our manual solution, a number of DSE limit cases are first chosen to give some time and space bounds to the DSE problem. Firstly, all of the DFGs for the kernels are separately spatially mapped to the PE array. This gives maximum possible concurrency, and is an upper bound on the size of the array. Typically, such a flat mapping results in a PE array that is too large, so DFGs need to be time-multiplexed. DFGs are considered singly, and in pairs (for DFGs that can be run concurrently) to find a suitable PE array size that can efficiently time-multiplex different DFGs and also fit in the available resources. Individual DFG operators need to be mapped to specific PEs. This is done so that, as far as possible, a specific PE is mapped to similar functions in each DFG embedding. More formally, we wish to choose a mapping which minimizes the total area of the PEs in the array (or maximizes the proportion of array logic that can be deleted), while preserving the same overall execution time as if fully functional PEs were used.

The final PE array then consists of a heterogeneous array of PEs whose individual capabilities are a superset of the capabilities that the PE requires for each DFG mapping, but which do not have functions that are never used for any application. Unused communications links and local memories can also be removed. A final check is made that this superset-optimized PE array fits within the available resources.

PE Layout Synthesis. Once the PE array layout has been finalized, the next step is to describe the PE array as VHDL code. This process is tedious and error prone because any error in the port mapping can lead to wrong results. To make the design flow a generic one without requiring any HDL programming from the user, a Perl script was created. This Perl script takes the information about the PE structure as a text file. The text file also contains information about the numeric capability of each PE such as the fixed and floating point support, the mathematical operation that each PE is supposed to perform over the period of multiple kernel executions, the data bit width and the presence of local memories. Using this information the Perl script automatically generates the VHDL code for the PE array. The script automatically selects the right kind of PE from a library of predefined PEs. The interface between the Microblaze processor and the PE array is already defined and needs no change. So the complete VHDL description of the system can be generated. The hardware description of QUKU, consisting of the Microblaze and the PE array is synthesized using the standard Xilinx FPGA tool flow. After the synthesis and backend flow is over, a bit-stream is generated consisting of the placed and routed design containing the Microblaze processor, the PE array and other peripherals. The software executable file generated for the Microblaze is loaded into the Microblaze system memory, which may be on-chip FPGA memory or external DRAM.

4. EXAMPLE APPLICATION: EDGE DETECTION

Edge detection is a common operation in image and video processing, as one step in an image recognition pipeline. The best edge-detection algorithm for a camera may depend on the lighting and noise conditions at the time, for example surveillance images during daytime and nighttime may work better with different algorithms. This application considers a system that implements two different edge-detection algorithms simultaneously: a Sobel edge detector, and a Laplace second gradient derivative edge-detector [Acharya and Ray 2005]. More details of the algorithm implementations can be found at [Shukla 2008]. This experiment considers a number of different FPGA implementation strategies for this application.

In this experiment, the Sobel and Laplace edge detection kernels are implemented on an FPGA in four different ways: QUKU, custom hardware circuit, a fixed 4 × 4

PE, and as software running on the Microblaze. The experiments used a Xilinx ML401 development board, which contains a Virtex-4 LX25 FPGA, plus external SRAM and DRAM chips. A camera records images which are copied onto a desktop machine. The FPGA board is connected to this desktop machine by an Ethernet link. A minimal functionality web server is implemented on the FPGA to send and receive TCP/IP packets. The desktop machine can access the main webpage which is hosted on the FPGA based web server and select an image on which the edge detection operation has to be done. The hardware processes the raw image and sends it to the desktop client for display.

The hardware consists of a Microblaze soft processor plus an optional hardware accelerator. This accelerator is either QUKU, or a fixed 4 × 4PE array, or a custom hardware core designed with System Generator for DSP. The Microblaze is responsible for general house-keeping functions and controlling the execution on the hardware accelerator while the hardware accelerator executes the computationally intensive part of the algorithm to enhance the overall performance of the system. Images are stored in a local file system implemented on the FPGA development board's SRAM. This SRAM has a size of 1Mbyte, and so the maximum available image size is 512 × 512 pixels, since original image, and result images after Sobel and Laplace edge-detection are needed. (In our simple implementation all original and result images are stored at 8 bits per pixel without image overwriting.)

4.1. Edge Detection implementation on QUKU

The data flow diagrams of the Sobel and Laplace kernels are shown in Figure 7. The data flow graphs show the pipelining and time steps required to complete one pixel gradient calculation. Here, pipeline refers to the flow of data through PEs and not the register level pipelining. The Sobel kernel requires 5 pipeline stages while the Laplace kernel requires 4 pipeline stages. The operation required by each PE is also shown in the circle.

Once the data flow graph of each kernel is prepared, the next step is to generate a PE network diagram that can execute both kernels. The merging of individual data flow graphs is done manually for this first version of QUKU. The merging algorithm used in this case is very simple – each node in the smaller graph (Laplacian) is mapped to a corresponding node in the larger graph (Sobel) which has the best match in terms of connectivity and function. In this case, the Laplacian algorithm can be totally embedded with the Sobel graph. [Moreano et al. 2005] present a DFG merging algorithm that could form the basis of an automated graph merging algorithm for QUKU.

Figure 8 shows the data flow graph obtained after merging the two data flow graphs in Figure 7. The circular PEs and single lined interconnect in the resultant DFG are common to both the Sobel and Laplace kernels while dashed interconnects indicates Laplace only and squared PEs and double lined interconnects indicate Sobel only parts of the system.

The merged data flow graph is taken as the final PE array layout to be implemented in hardware. This PE layout is heterogeneous in nature. The PEs have only the arithmetic capability that they actually need and the interconnections are also customized.

4.2. Edge Detection Implementation on a Fixed PE Array

In an ASIC CGRA, the physical characteristics like size of array, interconnect network, and PEs capabilities are fixed. To demonstrate the effectiveness of QUKUs per-PE customization, a homogeneous array of PEs is built. Each PE is able to execute all of the functions for the DFG, and all local interconnections are present, even if not used for this application. The fixed PE array consists of 16 PEs arranged in a 4 × 4 array. Each PE is connected to its four immediate neighbouring PEs with point-to-point links.

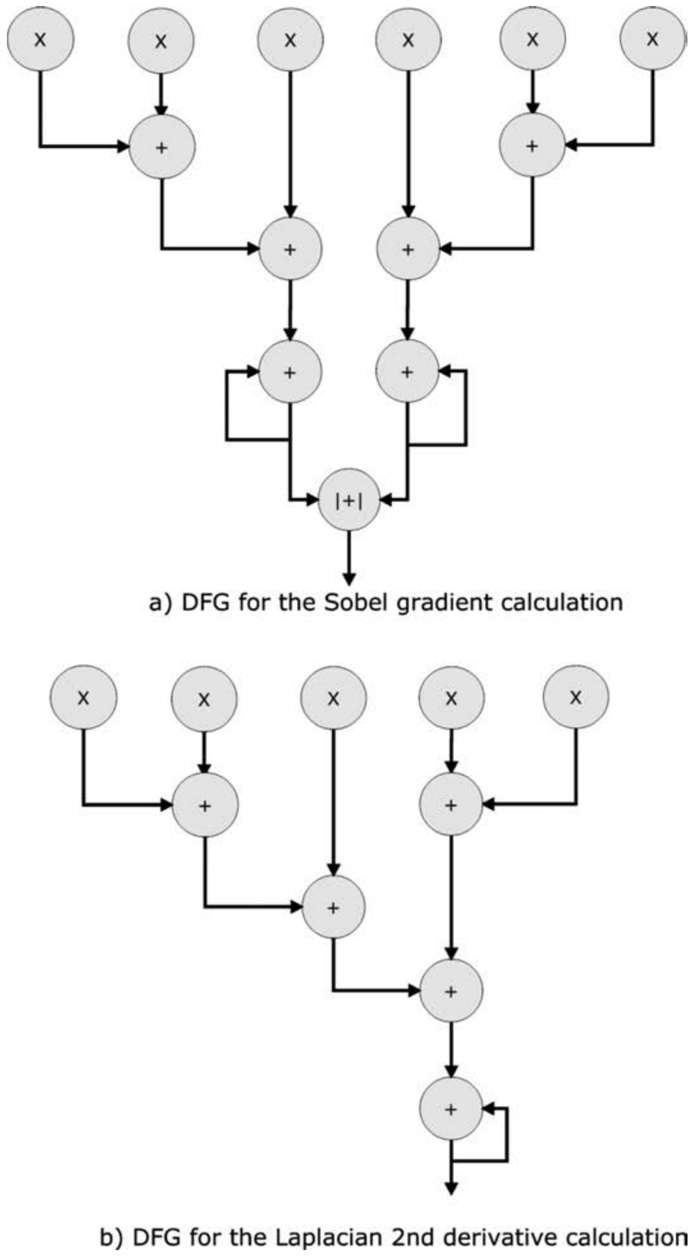

X
+
|+|
a) DFG for the Sobel gradient calculation
b) DFG for the Laplacian 2nd derivative calculation

Fig. 7. Data flow diagram for Sobel and Laplacian kernel.

The Fixed PE array is basically the same array as QUKU, but the per-PE optimizations of removing unused operators and communication links has not been done.

4.3. Edge Detection Implementation as Custom Circuit

To compare the performance of QUKU with a custom accelerator approach, the Sobel and Laplace edge detection kernels are implemented as separate IP cores in the FPGA,

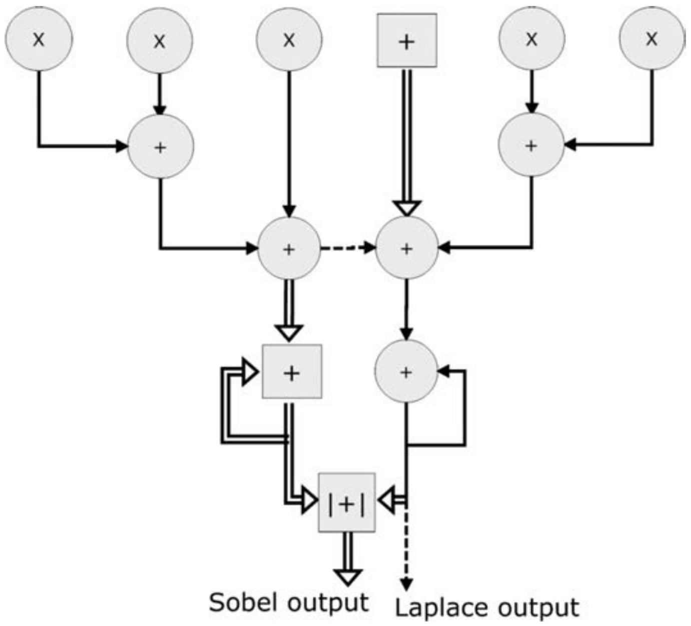


Fig. 8. Merged dataflow graph for the Sobel and Laplace kernels.
Key: Circular operators common to both
Square operators: Sobel only
Single lines: Common to both
Double Lines: Sobel only
Dashed Line: Laplacian only

designed using System Generator for DSP (based on the DFGs presented earlier in Figure 8), and connected to the processor with Fast Simplex Links [Rosinge 2004]. The DFGs presented above are implemented as a fixed network of arithmetic operators. The FPGA is not large enough to fit both cores at the same time, so swapping between cores requires reconfiguration of the FPGA.

4.4. Results and Analysis

This section describes the area, processing time and energy consumption for all the hardware and software implementations. Table I summarizes the measured and calculated results. Explanations of how each measurement was made are given below.

Area. The hardware implementations of QUKU, 4 × 4 PE array, custom IP based implementation and Microblaze-only implementation were done on the same ML401 development board. The percentage slice count is calculated as a percentage of the maximum slices available in the XC4VLX25 device. The custom IP based implementation of both detectors, and 4 × 4 PE array based implementations were too large to fit into the XC4VLX25 FPGA (>100% slice occupancy). To evaluate the performance in the custom IP based circuit, implementation was done in two steps; containing only one IP at a time. First the hardware was implemented with only the Sobel circuit and all the results were generated. Then in the second implementation, Laplace circuit was built and the results were generated. However, the area is calculated as if both cores were present. For the 4 × 4 PE array, some PEs were stripped of just enough unused local memories and arithmetic capabilities to fit on the FPGA. The original size of 128% of resources is listed, and used later in area-time calculations (even though it was trimmed to fit into 100% for the time measurements). Execution time was not

Table I. Results

	QUKU	4 × 4 PE Array	Custom Circuit	Software (Microblaze)
Area				
Slices (% of one FPGA)	9603 (89%)	13,857 (128%)	11,180 (104%)	4341 (40%)
BRAMs	54	89	38	12
DSP48 Blocks	9	19	8	3
Configuration Size in bytes (partial reconfiguration bitstream size for custom circuit)				
Sobel	38	78	454k	–
Laplacian	42	54	573k	–
Processing Time (Seconds) vs Image Size & Algorithm (S = Sobel, L = Laplacian) [Relative to QUKU]				
64 × 64 S	0.1215	0.122 [100.4%]	0.0898 [74%]	0.673 [553%]
128 × 128 S	0.3058	0.3077 [100.6%]	0.2663 [87%]	3.268 [1069%]
256 × 256 S	1.0243	1.0323 [100.8%]	0.9613 [94%]	26.48 [2585%]
512 × 512 S	3.9100	3.938 [100.7%]	3.7753 [96.6%]	331.2 [8470%]
64 × 64 L	0.093	0.0934 [100.4%]	0.0919 [98.8%]	0.9338 [1004%]
128 × 128 L	0.2778	0.2794 [100.6%]	0.2749 [98.9%]	5.2642 [1895%]
256 × 256 L	1.012	1.019 [100.7%]	0.9997 [98.8%]	44.276 [4375%]
512 × 512 L	3.953	3.96 [100.2%]	3.9011 [98.7%]	548.81 [13883%]
Area-Time Product (Slice-seconds) vs Image Size & Algorithm (S = Sobel, L = Laplacian) [Relative to QUKU]				
64 × 64 S	1166.76	1690.55 [145%]	1003.96 [86%]	2921.49 [250%]
128 × 128 S	2936.60	4263.80 [145%]	2977.23 [101%]	14186.39 [483%]
256 × 256 S	9836.35	14304.58 [145%]	10747.33 [109%]	114949.68 [1169%]
512 × 512 S	37547.73	54568.87 [145%]	42207.85 [112%]	1437739.20 [3829%]
64 × 64 L	893.08	1294.24 [145%]	1027.44 [115%]	4053.63 [454%]
128 × 128 L	2667.71	3871.65 [145%]	3073.38 [115%]	22851.89 [857%]
256 × 256 L	9718.24	14120.28 [145%]	11176.65 [115%]	192202.12 [1978%]
512 × 512 L	37960.66	54873.72 [145%]	43614.30 [115%]	2382384.21 [6276%]
Energy Consumption (mJ) vs Image Size & Algorithm (S = Sobel, L = Laplacian) [Relative to QUKU]				
64 × 64 S	0.283	0.303 [107%]	0.254 [90%]	1.3 [459%]
128 × 128 S	0.712	0.765 [107%]	0.754 [106%]	6.326 [888%]
256 × 256 S	2.384	2.567 [108%]	2.723 [114%]	51.26 [2150%]
512 × 512 S	9.1	9.794 [108%]	10.695 [118%]	641.2 [7046%]
64 × 64 L	0.216	0.232 [107%]	0.26 [120%]	1.8 [833%]
128 × 128 L	0.646	0.695 [108%]	0.778 [120%]	10.19 [1577%]
256 × 256 L	2.355	2.534 [108%]	2.832 [120%]	85.7 [3639%]
512 × 512 L	9.195	9.848 [107%]	11.05 [120%]	1062.5 [11554%]

affected by this circuit compression, however power figures for the custom circuit and for the fixed PE array will be slightly optimistic.

Processing Time. Processing time was measured by downloading the FPGA and CGRA configurations to the ML401 and measuring elapsed time, so it includes all aspects of the system operation, such as memory and I/O delays. Time for user-interface and web-server interaction was not included. The measured times include the following algorithm stages.

—Time spent in reading the raw image from the SRAM into the PE array local memory in case of QUKU and the 4 × 4 PE array or into the local memory of IP cores in case of the custom IP based implementation
—Algorithm execution time
—Time elapsed in writing the modified image into the SRAM from the local memory

The custom IP based implementation gives the best results among all the implementations while the Microblaze only implementation has the worst performance. The performance gap between the Microblaze only and other implementations widens exponentially as the image size increases, most likely due to the limited cache size of the Microblaze. The performance difference between the custom IP circuit, QUKU and 4 × 4 PE array is relatively small.

Area-time Product. It is often the case that speed performance improvement can be gained by adding additional circuitry that increases area. To understand if this extra area is worthwhile, a useful measure of the efficiency of an implementation is the area-time product. In this case, the area-time product is calculated by multiplying the FPGA slices used for each circuit by the execution time in seconds. QUKU gives the best area-time performance for all cases except the 64 × 64 Sobel test (most likely due to some initial delay associated with configuration). In all cases, the FPGA was globally clocked using a single common clock frequency of 100 MHz.

Energy Consumption. Another measure of the efficiency of an algorithm is the energy consumption, i.e., the power multiplied by the computation time. To calculate the energy consumption for each case, average power was calculated using the Xilinx CAD tool Xpower. Then the energy consumption was calculated using the power-energy relationship. QUKU gives the best energy performance for all cases except the 64 × 64 Sobel test. It appears that the cost of configuration setup for small kernels puts QUKU at a small disadvantage compared to a custom circuit.

So, in summary, the numbers above in Table I reveal the following key results. In terms of processing area, QUKU has slightly smaller area than a custom DSP circuit which can execute two different kernels. So a programmable architecture like QUKU is likely to have area benefits compared to custom circuits when different kernels need to be applied at different times. The ability to remove parts of the general PE architecture which are unused for a particular set of applications provides a significant area saving, in this case more than 20%. In terms of processing time, QUKU gives similar results to custom circuits provided that the size of the data to be processed is sufficient to overcome the setup overheads. Finally, QUKU provides the lowest energy consumption of all the tested options for all except the smallest data sets.

4.5. A Comment about Runtime Reconfiguration

The results above presented the area and processing time for IP based custom circuit implementation without using partial reconfiguration. It was shown that the custom IP based implementation consisting of both the IP cores was too big to fit in the chosen FPGA. Due to the larger area, the custom circuit implementation had a larger area-time product. An argument can be made that the Sobel and Laplace IP cores could be loaded at different instances in the FPGA using partial runtime reconfiguration thus conserving area, and improving the area-time product.

Run-time partial reconfiguration can be done using either module-based or difference-based partial reconfiguration techniques [Eto 2007]. In this experiment, the difference based partial reconfiguration technique has been used. First of all, a design consisting of the processor sub-system, which is static part of the system, and the Laplace IP core are synthesized and the full bitstream is generated. The same is done for processor and Sobel IP core. Then, partial bitstreams, consisting of the Sobel IP core and Laplace IP core are generated. The difference based partial bitstream consists of only those frames that are different between the two implementations. Assuming the system is initially loaded with the Laplace circuit, there are 3 bitstreams, a full bitstream consisting of the static part of the design and the Laplace IP core, a partial bitstream for the Sobel IP core and another partial bitstream consisting of the Laplace IP core. The size of the partial bitstreams for the Sobel and Laplace IP cores are 454 KB and 573 KB respectively which is 47% and 60% respectively of the full bitstream size for the Virtex-4 LX25 device. The partial reconfiguration times for Sobel and Laplace IP cores, using the Master-serial configuration method, are just the percentage size of the partial bitstream times the full configuration time, 62ms and 78 ms respectively.

An argument can be made that dynamic partial reconfiguration in an FPGA is analogous to the coarse grained reconfiguration in QUKU because in both the cases the two kernels are time multiplexed to achieve the desired result. The QUKU reconfiguration code for the Sobel and Laplace filters are 608 bits, and 672 bits respectively, giving reconfiguration times of 380ns and 420ns.

In QUKU, the coarse grained reconfiguration is used to load the Sobel and Laplace kernels on the PE array and in the custom circuit based implementation the partial bitstream technique could be used at run-time to load the Sobel and Laplace cores. The configuration code size in QUKU as compared to the partial bitstream size for the Sobel and Laplace kernels is 0.016% and 0.014% respectively. The configuration time to load the Sobel and Laplace kernels in QUKU is 0.61% and 0.54% of the partial reconfiguration time for the custom circuits. CGRA-style reconfiguration can give much faster configuration, and is considerably easier to manage than partial dynamic reconfiguration.

5. CRITICAL ANALYSIS AND CONCLUSIONS

As with any architecture, QUKU has its own advantages and disadvantages. This section analyzes the merits and demerits of QUKU with particular reference to the application described above.

With the FPGA used, the QUKU PE array and a Microblaze can fit on the FPGA. The PE array is approximately the same the size of the Microblaze, meaning that a typical QUKU system is approximately twice the size of a Microblaze alone. In terms of performance, speedups of between 5 and 138 times compared to Microblaze were achieved for different algorithms. Area-time product shows improvements of 2.5 times (for small image processing problems) up to 60 times (for large images) compared to Microblaze.

Compared to a fixed PE array, customization of the QUKU array provides considerable savings in area and area-time product (around 40%) since unused components are not loaded onto the FPGA. In comparison to custom FPGA circuits, QUKU has demonstrated comparable speed performance (in most cases within 5% of the custom circuit), and superior area-time product due to better operator re-use and scheduling. Furthermore, QUKU has shown the best energy performance of all the implementation options explored for all but the simplest filter tested. As the filter size increases, the relative cost of loading operands and retrieving results compared to executing kernels is reduced, and QUKU's relative advantages over a custom hardware solution are improved. This suggests that QUKU is more suited to applications with a higher ratio of computation versus communication with the host. In terms of reconfiguration, QUKU level reconfiguration involves reconfiguration code sizes and reconfiguration times more than 1000 times smaller than dynamic partial reconfiguration of the FPGA bit stream.

It is likely that for any particular kernel, a custom hardware implementation of that kernel could be designed that is more energy and area efficient than QUKU. However, when multiple kernels need to be swapped in real-time, the very slow reconfiguration at FPGA-level swings the balance back in favour of a coarse-grained kernel such as QUKU.

QUKU has been used to demonstrate the benefits of two technical innovations – the use of CGRA-like PE arrays as a viable computing architecture for FPGA application acceleration; and the benefits of per-PE customization to minimize overall array resource use. In both cases, the experimental circuit implementations have confirmed these benefits.

QUKU has been explored as a design method for an FPGA stream-based processing kernel, as part of larger stream-based applications. Like all FPGA-based kernel accelerators, care needs to be taken to ensure that kernel accelerations can be translated into application accelerations, and that data transfer between host and accelerator are able to usefully employ that kernel acceleration.

There is still much work that is needed to turn the ideas in QUKU into a viable design methodology. One key task that would be needed is the development of an accompanying set of CAD tools. Tools for profiling, mapping and scheduling, and PE customization would all need to be developed. Additionally, more work would be needed on the QUKU architecture, particularly on the I/O subsystem. This first prototype of QUKU used a common clock for the processor and the PE array. Synthesizing the array separately demonstrated that it could run at 175 MHz compared to the processor's 100 MHz, so a relatively simple improvement to QUKU that could yield immediate benefits would be to operate the system with different clock rates for processor and PE array.

Another task would be to define the range of stream-based applications that would map well to QUKU. For example, QUKU is based on PEs which have a consistent wordlength (such as 8 or 16 bits) throughout the system. It is unlikely that such an array would be suitable for variable wordlength applications, or for bit-level applications such as one would find in Huffman or Arithmetic Coding.

Overall, as a proof of concept, QUKU has successfully demonstrated that QUKU style CGRA architectures are an excellent approach to quickly harnessing the power of FPGA reconfigurability to customize architectures to applications. At the same time, CGRA-level high-speed configurability provides an attractive approach to hardware re-use within an application without the difficulties and delays involved in partial dynamic FPGA reconfiguration.

REFERENCES

ACHARYA, T. AND RAY, A. K. 2005. *Image Processing: Principles and Applications*. John Wiley and Sons, Hoboken NJ.

AHN, M., YOON, J. W., PAEK, Y., KIM, Y., KIEMB, M., AND CHOI, K. 2006. A spatial mapping algorithm for heterogeneous coarse-grained recodigurable architechures. In *Proceedings of the Conference on Design, Automation and Test in Euope (DATE'06).* European Design and Automation Association, Leuven, Belgium, 363–368.

BAAS, B., YU, Z., MEEUWSEN, M. SATTARI, O. APPERSON, R. WORK, E. WEBB, J., LAI, M., MOHSENIN, T., TRUONG, D., AND CHEUNG, J. 2007. AsAP: A fine-grained many-core platform for DSP Applications. *IEEE Micro 27,* 2, 34–43.

BAUMGARTE, V., MAY, F., NÜCKEL, A., VORBACH, M. AND WEINHARDT, M. 2003. PACT XPP - A Self-Reconfigurable Data Processing Architecture. *J. Supercomput. 26,* 2, 167–184.

CALLAHAN, T. J. 2002. Compilation of C for hybrid reconfigurable architectur. PhD Thesis, Computer Science Department University of California, Berkeley, CA.

ETO, E. 2007. Difference-based partial reconfiguration. Xilinx Appli. Note. www.xilinx.com.

FRIEDMAN, S., CARROLL, A., VAN ESSEN, B., YLVISAKER, B., EBELING, C., AND HAUCK, S. 2009. SPR: An architecture-adaptive CGRA mapping tool. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '09)*. ACM, New York, NY, 191–200.

HAUSER, J. R. AND WAWRZYNEK, J. 1999. Garp: a MIPS processor with a reconfigurable coprocessor. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. K. L. Pocek, J. M. Arnold, Eds., IEEE Computer Society, Los Alamitos, CA, 12–21.

HEYSTERS, P. M. 2002. Coarse-grained reconfigurable processors - flexibility meets efficiency. PhD Thesis University of Twente, The Netherlands.

HWANG, J., MILNE, B., SHIRAZI, N., AND STROOMER, J. D. 2001. System level tools for DSP in FPGAs. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. G. J. Brebner and R. Woods, Eds., Springer, Berlin, Germany, 534–543.

LANGE, S. AND MIDDENDORF, M. 2008. Design aspects of multi-level reconfigurable architectur. *J. Signal Proce. Syst. 51,* 7, 23–37.

LEE, M., SINGH, H., LU, G., BAGHERZADEH, N., AND KURDAHI, FJ. 2000. Design and implementation of the MorphoSys reconfigurable computing processor. *J. VLSI Signal Process. Syst. Signal, Image and Video Technol. 24,* 2-3, 147–164.

LYSAGHT, P., BLODGET, B., MASON, J., YOUNG, J., AND BRIDGFORD, B. 2006. Enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAs. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. Spain, A. Koch, P. Leong, and E. Boemo, Eds., IEEE, 1–6.

MASTER, P. 2002. The age of adaptive computing is here. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. M. Glesner, P. Zipf, and M. Renovell, Eds., Springer, Berlin, Germany, 1–3.

MEI, B., VERNALDE, S., VERKEST, D., MAN, H. D., AND LAUWEREINS, R. 2003. ADRES: An Architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. P. Y. K. Cheung, G. A. Constantinides, and J. T. De Sousa, Eds., Springer, Berlin, Germany, 61–70.

MOREANO, N., BORIN, E., CID DE SOUZA, AND ARAUJO, G. 2005. Efficient datapath merging for partially reconfigurable architectures. *IEEE Trans. Comput. Aid. Des. Integrat Circuits Syst. 24*, 7, 969–980.

OPPOLD, T., SCHWEIZER, T., FILHO, J. O., EISENHARDT, S., AND ROSENTHAL, W. 2007. CRC – Concepts and evaluation of processor-like reconfigurable architectures. *Inf. Technol. 49,* 3, 157–164.

RECORE SYSTEMS. 2007. Montium reconfigurable digital signal processing tile processor datasheet, www.recoresystems.com.

ROSINGE, H.-P. 2004. Connecting customized IP to the MicroBlaze soft processor using the fast simplex link (FSL) Channel. Xilinx Appli. Note XAPP529. http://www.origin.xilinx.com/support/documentation/application_notes/xapp529.pdf.

SHUKLA, S. K. 2008. QUKU: A mixed grain dynamically reconfigurable architecture for high performance computing. PhD Thesis, University of Queensland, Brisbane, Australia.

SINGH, H., LEE, M., LU, G., KURDAHI, FJ., BAGHERZADEH, N., AND FILHO, E. M. C. 2000. MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Comput. 49,* 5, 465–481.

SMIT, G. J. M., GUO, Y., AND HEYSTERS, P. M. 2004. Overview of the tool-flow for the Montium processor tile, In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms* T. Plaaks, Ed., CSREA Press, Irvine, CA, 45–51.

TESSIER, R. AND BURLESON, W. 2001. Reconfigurable computing for digital signal processing: A survey. *J. VLSI Signal Proces. 28*, 1-2, 7–27.

TODMAN, T. J., CONSTANTINIDES, G. A., WILTON, S. J. E., MENCER, O., LUK, W., AND CHEUNG, P. Y. K. 2005. Reconfigurable computing: architectures and design methods, *IEE Proc. Computers and Digital Techniques 152*, 2, 193–207

TOI, T., NAKAMURA, N., KATO, Y., AWASHIMA, T., WAKABAYASHI, K., AND JING, L. 2006. High-level synthesis challenges and solutions for a dynamically reconfigurable processor. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '06)*. ACM, New York, NY, USA, 702–708.

VAN ESSEN, B. C., PANDA, R., WOOD, A., EBELING, C., AND HAUCK, S. 2011. Energy-efficient specialization of functional units in a coarse-grained reconfigurable array. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11)*. New York, NY, 107–110.

WESTCOTT, T. AND DOCEF, A. 2007. Optimi-Ware: Software profiling and optimization [Best of theWeb]. *IEEE Signal Proces. Maga. 24*, 131–133.

WOODS, R., MCALLISTER, J., TURNER, R., YI, Y., AND LIGHTBODY, G. 2008. *FPGA-Based Implementation of Signal Processing Systems.* Wiley, Chichester, UK.

XILINX. 2007. Microblaze Processor Reference Guide v7.1, www.xilinx.com.

Yoon, J. W., Shrivastava, A., Park, S., Ahn, M., Jeyapaul, R., and Paek, Y. 2008. SPKM: a novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '08)*. IEEE Computer Society Press, Los Alamitos, CA, 776–782.

Received December 2010; revised April 2011; accepted August 2011