A Dynamically Reconfigurable Asynchronous Processor For Low Power Applications

K.A. Fawaz, T. Arslan, S. Khawam, M. Muir, I. Nousias, I. Lindsay, A. Erdogan

School of Engineering
University of Edinburgh, The King's Buildings, Mayfield Road, Edinburgh, EH9 3JL, UK
K.Fawaz@ed.ac.uk

ABSTRACT

There is an increasing demand in high-throughput mobile applications for programmability and energy efficiency. Conventional mobile Central Processing Units (CPUs) and Very Long Instruction Word (VLIW) processors cannot meet these demands. In this paper, we present a novel dynamically reconfigurable processor that targets these requirements. The processor consists of a heterogeneous array of coarse grain asynchronous cells. The architecture maintains most of the benefits of custom asynchronous design, while also providing programmability via conventional high-level languages. When compared to an equivalent synchronous design, our processor results in a power reduction of up to 18%. Additionally, our processor delivers considerably lower power consumption when compared to a market leading VLIW and a low-power ARM processor, while maintaining their throughput performance. Our processor resulted in a reduction in power consumption over the ARM7 processor of around 9.5 times when running the bilinear demosaicing algorithm at the same throughput.

Index Terms— Asynchronous logic circuits, Reconfigurable architectures

1. INTRODUCTION

Current architectures will not be able to cope with the increasing demand of high-throughput mobile applications for better programmability and energy efficiency. Typical examples are mobile devices for next generation networks where a high amount of image and signal processing is required. Conventional mobile Central Processing Units (CPUs) cannot meet the throughput demand, forcing manufacturers to rely on custom hardware accelerators. This sacrifices programmability, leading to increased product lead-time and risk. Very Long Instruction Word (VLIW) processors can offer an increase in performance over most superscalar CPUs by taking advantage of instruction-level parallelism. However, this increase is at the expense of high power consumption and greater compiler complexity. Reconfigurable datapaths offer better solutions when power and area considerations are also taken into account for high throughput applications (streaming). The two main categories of reconfigurable datapaths are Fine-grain and Coarse-grain architectures. Most mobile application algorithms require datapaths one word in size. As a result, coarse-grained reconfigurable computers are more suited for such applications than fine-grained ones. There are several available coarse-grain architecture designs [1]. Despite providing good results in high computational performance and flexibility, they typically either do not provide enough power savings or are too difficult to program.

1.1. First step – RICA

A novel clocked reconfigurable datapath solution called RICA has been successfully demonstrated at the University of Edinburgh. The processor is programmed via conventional high-level software languages like C [2]. This allows existing code bases and skills to be leveraged, and satisfies the demand for programmability in mobile applications. RICA consists of an array of coarse-grain customisable cells that can be dynamically reconfigured on every clock cycle. RICA also controls its own reconfiguration. In programmable logic devices, timing requirements change depending on what datapath is being mapped and the level of pipelining required. In most programmable devices, the maximum operating frequency is limited to the largest critical-path delay of all the steps of the program being mapped. However, RICA improves the operating frequency of a loaded program by estimating the worst case timing of each mapped datapath in software and programming it as part of the array's configuration. A programmable clock frequency is achieved by dividing the main clock by an amount specified in the configuration. The drawback of this approach is that it leads to complex clock hardware. Additionally, there is always some idle time whilst waiting for the last clock cycle to end.

1.2. Next step – Asynchronous

Asynchronous logic is a method of designing digital systems without clocks. Global synchrony is replaced with local synchronisation amongst parts that exchange data. Local synchronisation is achieved through the use of local request (req) and acknowledge (ack) signaling called handshakes [3]. The handshaking protocol implements communication and synchronisation among the components of an asynchronous datapath irrespective of its length.

In this paper, we present and evaluate our novel dynamically reconfigurable asynchronous processor (DRAP) aimed at high-throughput mobile applications. By basing our architecture on RICA and using asynchronous design techniques, we achieve lower power consumption than leading processors while maintaining a high level of programmability. The main distinguishing features of our reconfigurable architecture are as follows:

• **Fine-grain clock gating**: Asynchronous logic implements fine-grain clock gating by automatically turning off unused circuits. This results in event-driven energy consumption.

• **Inherent pipelining**: The asynchronous cells contain latches/flip-flops within them. As a result, a certain degree of inherent pipelining is provided for repetitive executions of datapaths.

• **Reduced software and hardware complexity**: Average-case communication between the array cells is automatically enforced by handshaking irrespective of the mapped datapath length. Therefore the need for both worst-case datapath timing estimation and logic to implement clock division (as in [2]) is eliminated.

• **Reduced program size**: There is no need to save clock information for every configuration step. Since asynchronous cells already contain latches within them, fewer pipelining dedicated registers than equivalent clocked designs are needed. This achieves further memory savings and also reduces the area of the design since fewer switches and their configuration bits are required. We estimated that our design would result in a 10-15% smaller program memory size than that of the reconfigurable synchronous architecture described in [2] (array of 500 operational cells and 2000 registers).

2. RELATED WORK

2.1. The RICA processor

RICA based processors are coarse grained reconfigurable computing fabrics, consisting of a heterogeneous array of programmable cells on a programmable interconnect network [2]. A diagram of a simplified RICA array is shown in Fig. 1. The operational cells are chosen to match the data width and functionality of RISC instructions in a typical C compiler. They can be combined through the reconfigurable interconnect network to perform more complex instructions in a single configuration context - called a step. A configuration step persists for the time to allow the sequence of connected cells which form the critical path to complete, and then the next configuration step is loaded. The main features of RICA are as follows:

• RICA uses the concept of distributed registers—a significant fraction of the instruction cells are registers.

• The array uses a Harvard memory architecture (to maximise bandwidth)—the program memory and data memory are separate. In many application domains, the array can also have special-purpose stream memories (line buffers) which further increase the on-chip bandwidth.

• The structure of the core allows complex datapaths to be constructed between the available operational cells.

• The array is in control of its own reconfiguration: the JUMP cell (Fig. 1) provides access to the program counter and allows a mapped datapath to influence program control flow [2].

• To account for the varying critical path of each configuration step, a reconfiguration rate controller (RRC) is used to control the length of time (number of master clock cycles) for which the configuration step persists. This value is stored as part of each configuration context.

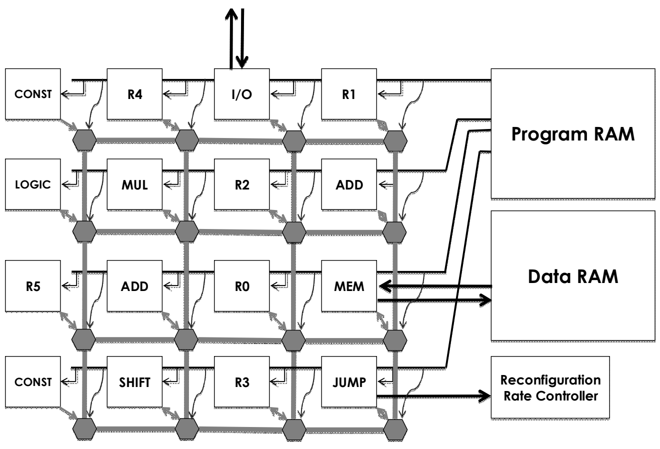


Fig. 1: Simplified example of a RICA based architecture.

2.2. Asynchronous reconfigurable processors

The vast majority of proposals for reconfigurable asynchronous processors target asynchronous fine-grain FPGAs. Early designs were based on modifying existing synchronous FPGA architectures by adjusting the functional units [4], adding reconfigurable delay lines [5], or replacing the clock by control signals generated by an array of timing cells [6]. Recent attempts aim to design fully asynchronous reconfigurable architectures. In [7] [8], asynchronous dataflow-based fine-grain FPGAs using finely pipelined dual-rail asynchronous circuits with four-phase handshaking are presented. In [9], an asynchronous reconfigurable architecture for cryptographic applications is presented; it uses homogenous coarse-grain cells with 8-bit wide data. This design is custom built for cryptographic applications and uses dualrail four-phase handshaking protocol. A hybrid architecture called Tartan is presented in [10]. It is composed of a hierarchical coarse-grained asynchronous

Reconfigurable Fabric (RF) and a RISC CPU core. Tartan uses the spatial computation model where applications developed in C language are compiled and translated by separate tools into RF logic netlist. The structure of the RF is based on the synchronous Piperench architecture [11] [12]. Tartan provides a successful example of applying asynchronous design techniques to a synchronous reconfigurable architecture to reduce energy consumption.

3. ASYNCHRONOUS DESIGN

This section gives background on commonly used asynchronous signaling protocols, data encoding schemes and delay models. Handshaking is the signaling scheme implemented by asynchronous circuits. The medium upon which two sub-systems communicate by handshaking is called a channel: this includes the control channel (request and acknowledge signals) and the data channel.

3.1. Signaling protocols

There are several choices of how to encode the alternating events of the request and acknowledge onto specific control wires. The two most pervasive signaling protocols are described below [13][14]:

• **Two-phase signaling**: In this protocol, each transition has a meaning, i.e. each transition on a wire signals a request or acknowledge. This means that two transitions are needed to complete a handshaking event.

• **Four-phase signaling**: This protocol uses the logical level of the request and acknowledge wires to control the handshake; to achieve this, both wires must return to logic zero at the end of a handshake. This means that four transitions are needed to complete an event.

A quick comparison between the two signaling protocols shows that four-phase leads to simpler and smaller hardware than two-phase as it should be sensitive to only one edge. It also allows more flexibility when transferring data. On the other hand, two-phase signaling is potentially faster and more power efficient than four-phase because it uses each transition in a handshake. However, this is mostly not the case as two-phase implementations require more logic complexity than equivalent four-phase ones; this increased logic complexity may consume more power than is saved by the reduced control transitions [15].

3.2. Data encoding

There are two options for how data is encoded in the data channel of an asynchronous circuit:

• **Bundled Data**: Data is encoded using one wire for each data bit and a separate wire to indicate validity of the entire data. The datapath in this case is much like a synchronous one. This approach is conservative in number of required wires; however, it relies on the timing assumption that the data is valid before the request signal is raised [13] [14].

• **N-of-M codes**: Data is encoded using M wires where only N of the M wires are ever active. No data valid signal is needed as validity is encoded within the M wires. This encoding scheme potentially leads to lower power consumption as fewer transitions on the wires occur compared to bundled data [13] [14] [16].

3.3. Delay models

Asynchronous circuits can be regarded as computing dynamically through time. Therefore a delay model is critical in defining the dynamic behaviour of such circuits. Delay models categorise circuits by the propagation delay assumptions of the circuit components. Such models provide a designer with a template for construction and verification of the circuit. The delay models of most interest for this paper are the Quasi Delay Insensitive (QDI) model and the self-timed model using matched delays. More information about the different asynchronous delay models can be found in [13][14]. Dual-rail encoding is often used in QDI designs where as bundled data is used with matched delay circuits.

4. SYSTEM ARCHITECTURE

DRAP consists of a heterogeneous array of course-grained asynchronous operational cells. Fig. 1 shows an abstract view of the architecture. The operational cells are designed using 4-phase handshaking protocol and bundled data encoding. Bundled data encoding was preferred over Quasi-delay insensitive encoding methods since it uses fewer wires and would result in smaller cells. The operational cells are interconnected through a network of programmable switches to allow the creation of datapaths. In a similar way to a CPU architecture, the configuration of the operational cells and interconnects are changeable to execute different blocks of instructions. The program memory contains the configuration bits that control both the ICs and the interconnect switches. Special cells in the core provide the interface to the data and program memories.

4.1. Design of operational cells

Each operational cell is limited to a small number of operations such as addition, multiplication, and logic operations. Synthesis of the cells was done using the automated decomposition tool Tide from Handshake Solutions [17]. A high-level concurrent programming language called Haste described the operational cells and then synthesised in two stages to a Verilog netlist based on cells from a standard-cell library. The first stage translates the Haste code into an intermediate Handshake Circuit in a transparent, syntax-directed process and the next stage maps the Handshake Circuit to a structural Verilog netlist and for initial circuit-level optimisation [17]. The resulting cell includes two parts (Fig. 2): the datapath which contains flip-flops, latches and combinatorial logic blocks, and the control which contains matched delay chains and asynchronous logic with feedback loops.

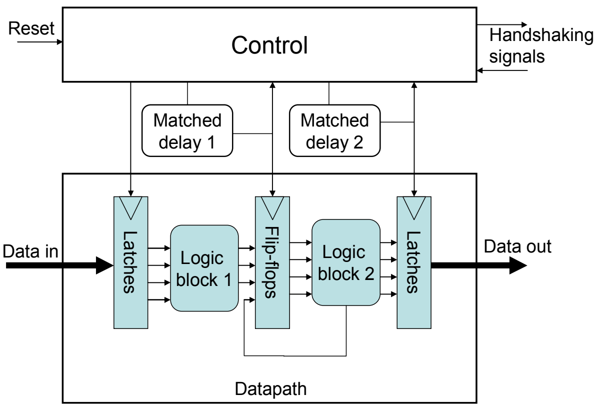
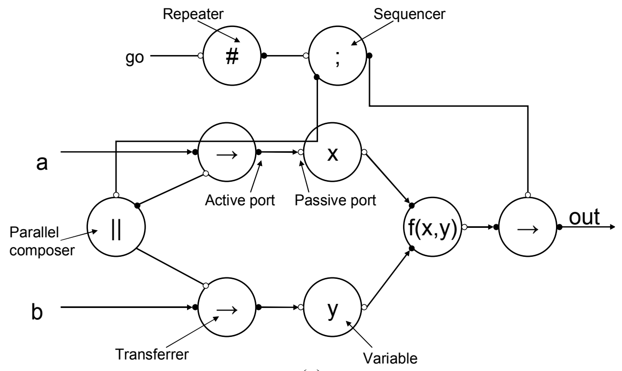
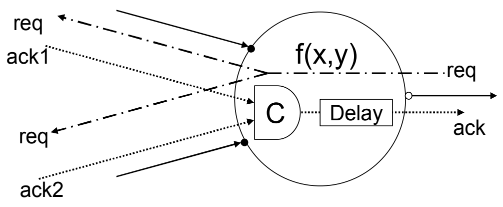


Fig. 2: Asynchronous circuit block diagram with the Control and Datapath.



(a)



(b)

Fig. 3: (a) Equivalent handshake circuit. (b) A closer look at control signals.

Fig. 3 shows the handshake circuit graph of a general 2-input, 1-output operational cell. When the cell is activated, a request signal is sent down both input channels *a* and *b* and data from the channels will only be transferred to the variables once both channels have acknowledged their respective requests. The gate labelled "C" is a C-element commonly found in asynchronous logic; it synchronises signals by firing a transition on its output only after each of its inputs have made transitions in the same direction. More information on the design of the operational cells their handshake circuits can be found in [18].

In conventional synchronous design, invalid data propagate through the inputs of cells and cause unnecessary activity. Asynchronous logic implicitly implements fine-grain "clock gating" by automatically turning off unused parts of the design. Additionally, each asynchronous cell will wait for the data at its input to become valid before letting it through. Hence unnecessary computation in used cells is eliminated and unused cells have no switching activity.

4.2. Design of interconnect structure

The general interconnect structure of an asynchronous reconfigurable architecture can be conceptually modeled after that of an equivalent synchronous architecture. However, asynchronous circuits require more wires than their synchronous counterparts to communicate information. In programmable asynchronous logic, a sender will communicate with any number of receivers depending on what is being programmed. The design of the interconnect must take into account the need for conditional acknowledge signal synchronisation. The traditional techniques to perform conditional acknowledge synchronisation, which are used in interconnect designs in [7] [8] [9], result in increased complexity and number of configuration bits compared to an equivalent interconnect design for synchronous communication. A novel method for conditional acknowledge synchronisation was developed by the authors in [19]. Our technique minimises control and configuration size compared to existing techniques.

Different solutions are available for the circuit design and for the topology of the switches, such as multiplexer-based crossbar or the island-style mesh found in typical FPGAs [20].

Several interconnect structures have been tested and compared. However, this comparison is beyond the scope of this paper. For our sample array, we modeled the interconnect design on the island-style structure. The configurable routing switches are built around the operational cell and allow each cell to communicate with its four nearest neighbors. The routing switches were designed to accommodate handshaking signals and perform conditional acknowledge synchronisation using our developed technique.

5. AUTOMATIC TOOL FLOW

An automatic tool flow has been developed in [2] for hardware generation and programming of synchronous coarse-grain arrays. The tools were extended to include support for asynchronous arrays. There are two main components of software support available for DRAP:

Array hardware generation: The tool takes a definition of the available operational cells in the array (types, count, and positions) along with other parameters such as interconnect bitwidth. The output is a synthesisable RTL definition of the array, which can be used in a standard system-on-chip (SoC) tool flow for verification, synthesis, layout, and analysis such as power consumption and timing.

Programming arrays from high-level languages (Fig. 4)**:** The granularity and self-reconfigurability of the DRAP architecture make it programmable in a broadly similar way to standard CPUs. This allows existing developments and methodologies such as optimising compilers to be used.

Step 1: High-Level Compiler: Takes the high-level code and transforms it into an intermediate assembly language. This step is performed by the industry standard open source GNU Compiler Collection (GCC) [21]. The resulting assembly describes the program as a series of basic blocks, each containing a list of instructions. Each instruction maps directly onto a DRAP asynchronous operational cell. Since GCC has grown up around CPU architectures, its output is based on the supposition that instructions are executed in sequence - i.e. one instruction per cycle; the compiler has no knowledge about the parallelism available on DRAP.

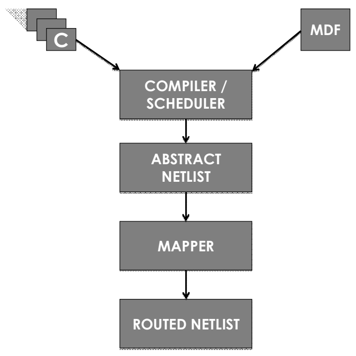


Fig. 4: Software flow for programming DRAP starting from high-level C program.

Step 2: DRAP Scheduling: In this step all the optimisations related to the DRAP architecture are performed. The DRAP scheduler takes the assembly output of GCC and creates a sequence of netlists representing the basic blocks of the program. Each netlist contains a group of instructions that will be executed on DRAP. Temporary registers allocated by GCC are replaced by simple wires. The partitioning is done according to dependencies between instructions: dependent instructions are connected in series, whilst independent ones are in parallel. The scheduling algorithm [22] takes into account the operational cell resources and timing constraints in the array to maximise cell occupancy and minimise the longest path delay.

Step 3: Allocate and Route: For each netlist in the program, the instructions are mapped to physical cells in the array. As there can be numerous available operational cells resources to which a given assembly instruction can be allocated, a tool is provided to minimise the distance between connected cells (similar to a standard place and route tool like VPR [23]).

Step 4: Configuration-Memory: From the mapped netlists, we can simply generate the required content of the configuration memory (program RAM).

If the required performance determined by RTL simulation is not met, then the high-level source code can be modified or the mixture of cell resources changed. Adjusting the hardware resources allows the architecture to be tailored to the specific application domain where it is to be used, thus saving area and power. Once array parameters have been decided upon, the generated files can be used for fabrication. If the algorithm continues to change during or after the fabrication process then the code is simply recompiled for those fixed resources.

Table I: Operational cells in sample array.

Cell	Count	Cell	Count
ADD/COMP	65	LOGIC	20
MUL	20	MUX	65
REG	134	MEM	4
SHIFT	35	JUMP	1
CONST	40	SBUF	16

Table II: Comparing DRAP with C_RICA, and ASIC.

Algorithm	**Through-put**	**DRAP**	**C_RICA**	**ASIC**
		Power (mw)		
Bilinear Demosaicing	23.6 MP/s	34.8	41	11.4
FFT	100 Mb/s	6.5	7.2	2.3
2D DCT	45 Mb/s	4.4	5.3	1.5
Viterbi	236 Mb/s	26.5	28.1	6.8

Table III: Comparing DRAP with ARM7, and TIC64.

Algorithm	**Through-put**	**DRAP**	**ARM7**	**TIC64**
		Power (mw)		
Bilinear Demosaicing	23.6 MP/s	34.8	326.4	408
FFT	100 Mb/s	6.5	52.6	132
2D DCT	45 Mb/s	4.4	31.8	57.2
Viterbi	236 Mb/s	26.5	162.9	285.2

6. EVALUATION

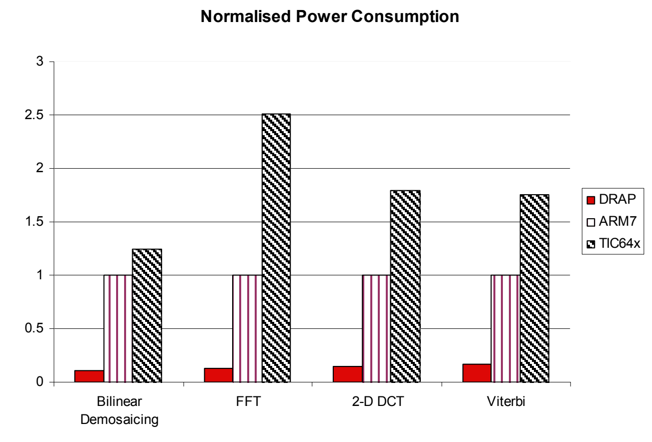
6.1. Sample design

A sample DRAP array was designed for comparison purposes. It contains 400 18-bit asynchronous cells as listed in Table I. These cells are interconnected using multiplexer-based switches. The mixture of the operational cells was manually selected to be adequate for general applications; other combinations can be chosen to be tailored to an application.

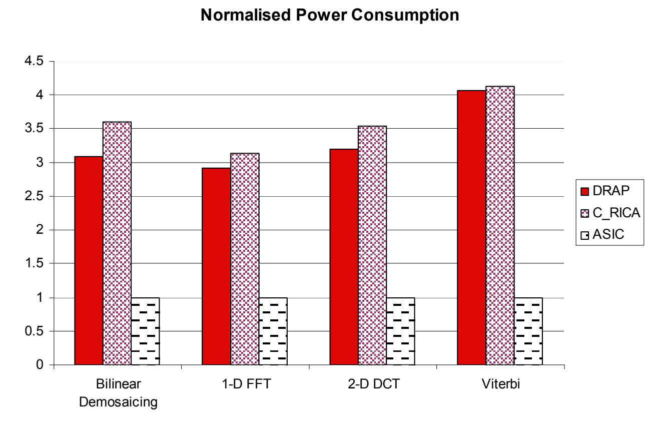
With the selected type of interconnects and operational cells, the reconfigurable core requires a total of 9260 configuration bits. The array was implemented using a UMC 0.13-µm technology. The sample DRAP was compared to an equivalent 400 cell array based on the RICA architecture in [2] (0.13-µm) referred to as

Custom_RICA_400 (C_RICA). It was also compared to an equivalent ASIC design of each of the tested algorithms (0.13-µm), the ARM7-TDMI-S [24] (0.13-µm) and the TIC64x 8-way VLIW [25].

For our evaluation, we selected sample algorithms representative of the more complex systems found in mobile and imaging applications: Bilinear demosaicing [26], 8K point radix-2 FFT [27], 2D DCT [28], and Viterbi [29]. All the benchmarks are direct unoptimised C representations of the algorithms—all optimisations are left for the C compilers (Level-3/O3). For each benchmark, the power consumption of each design was calculated for the same throughput.



(a)



(b)

Fig. 5: Normalised power consumption graph of the benchmarks on DRAP and other architectures. Power normalised with respect to (a) ARM7, (b) ASIC.

For the sample DRAP, C_RICA, and ASIC designs, the power and area were found using post-layout simulations on PrimePower from Synopsys (and post clock tree synthesis for the synchronous designs). The ARM7 datasheet [24] provides power and area value of the ARM core in 0.13-µm technology, while [30] allows us to estimate the power consumption of just the datapaths in the TI C64x. All these power estimations were measured at 1.2-V operating voltage and only focus on the energy consumed in the data path without the memory.

6.2. Results

The results are listed in Tables II and III and Fig. 5. From the table, we can see that for all the benchmarks we achieve better performance on the DRAP than on the conventional ARM7 CPU. The sample DRAP consumes around 6–10 times less power than the ARM7. However, it should be pointed out that the proposed DRAP is capable of achieving a much higher throughput performance than ARM7. When compared to the C64x VLIW, DRAP achieves a reduction in power consumption of 10–21 times. A big part of the power reductions achieved over the four DSP systems are savings gained by eliminating the register files and having distributed registers. Compared to an equivalent ASIC design of each algorithm, DRAP consumes only between 2.8-3.9 times more power.

To evaluate the benefit of using an asynchronous substrate, we compared DRAP to the equivalent C_RICA array. Our design achieved a power consumption reduction of up to 18% for the bilinear demosaicing algorithm when running at the same throughput. This is a direct result of the lower level of switching power in the asynchronous design due to its inherent fine-grain clock gating. The reduction in power comes at a cost of around 8% increase in area. Additionally, due to the added delay introduced by the 4-phase handshaking, it is estimated that the average throughput of DRAP can be up to 10% less than the maximum throughput which can be achieved by C_RICA.

7. CONCLUSION

In this paper, we presented a novel asynchronous coarse-grain reconfigurable processor. DRAP was designed to be energy efficient and programmable using high-level languages in order to target high-throughput mobile applications. By basing DRAP on the RICA architecture described in [2], we built a processor which can be easily reprogrammed through high-level languages. Energy efficiency was achieved by applying asynchronous design techniques on the operational cells and interconnect structure, leading to implicit fine-grain clock gating. This results in a lower level of switching power for datapaths which persist for a large number of loop iterations. The handshaking protocol of asynchronous design implements communication and synchronisation among the components of a datapath irrespective of its length. This allows DRAP to map different datapath lengths without the need for complex clocking schemes. Consequently, the hybrid software and hardware solution used in clocked arrays was eliminated, along with its contribution to configuration size. We estimated that our design would result in a 10-15% smaller program memory size than that of the reconfigurable synchronous architecture presented in [2] (for an array of 500 operational cells and 2000 registers).

Our results show that DRAP offers a significant reduction in power consumption compared to leading processors. DRAP outperforms ARM7 and the TI C64x VLIW processors by providing 6–10 times and 10–21 times less power consumption for a given throughput, respectively. Compared to an equivalent synchronous design based on [2], DRAP results in up to 18% reduction in power consumption when running the same algorithm at the same throughput. This comes at a cost of 8% increase in area. It is also estimated that there may be up to 10% reduction in maximum achievable throughput.

8. FURTHER WORK

The initial performance results of the asynchronous array are encouraging. More changes can be done to improve performance further: we aim to explore other interconnect schemes, since only a small realm of possible architectures have been examined - the island-style one. We also plan to introduce pipelining at the interconnect level and also within some operational cells in an effort to improve throughput. As a further step, we plan to investigate using other data encoding schemes and asynchronous techniques on the proposed design. Finally, we have identified the need to test the arrays in various ways: performing simulations on the logic array and memory, and also quantifying the benefits of the arrays in terms of storage requirements to these programs.

9. ACKNOWLEDGEMENT

This research is sponsored by the Overseas Research Student Awards Scheme (ORSAS) and by Edinburgh University's Institute for Integrated Micro and Nano Systems (IMNS).

10. REFERENCES

[1] Zain-ul-Abdin and B. Svensson, "Evolution in Architectures and Programming Methodologies of Reconfigurable Computing", Microprocessors and Microsystems, 2008.

[2] S. Khawam, I. Nousias, M. Milward, Y. Yi, M. Muir, and T. Arslan, "The Reconfigurable Instruction Cell Array," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 16, no. 1, 2008.

[3] R. Manohar, "Reconfigurable Asynchronous Logic," In IEEE Custom Integrated Circuits Conference, 2006, CICC '06, pp. 13-20, 2006.

[4] S. Hauck, S. Burns, G. Borriello, and C. Ebeling, "An FPGA for Implementing Asynchronous Circuits," IEEE Design and Test of Computers, vol. 11, no. 3, pp. 60-69, 1994.

[5] K. Maheswaran, "Implementing Self-Timed Circuits in Field Programmable Gate Arrays," master's thesis, Univ. of California Davis, 1995.

[6] R. Payne, "Asynchronous FPGA Architectures," IEE Computers and Digital Techniques, vol. 143, no. 5, 1996.

[7] J. Teifel and R. Manohar, "An Asynchronous Dataflow FPGA Architecture," IEEE Transactions on Computers, vol. 53, no. 11, pp. 1376–1392, 2004.

[8] C. G. Wong, A. J. Martin, and P. Thomas, "An Architecture for Asynchronous FPGAs", In IEEE International Conference on Field-Programmable Technology, 2003.

[9] K. Sun, X. Pan, and J. Wang, "Design of a Novel Asynchronous Reconfigurable Architecture for Cryptographic Applications," IEE International Multi-Symposiums on Computer and Computational Sciences, IMSCCS 2006, pp. 751-757, 2006.

[10] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, M. Budiu, and S. C. Goldstein, "Tartan: Evaluating spatial computation for whole program", ASPLOS'06, October 2006.

[11] S. C. Goldstein, H. Schmit, et al. PipeRench: a coprocessor for streaming multimedia acceleration. In International Symposium on Computer Architecture (ISCA), pages 28–39, May 1999.

[12] T. Bjerregaard, J. Sparsø, "A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip", In Proceedings of the Design, Automation and Test in Europe Conference and Exhibitions (DATE'05), Vol. 2, March 2005, pp. 1226-1231.

[13] A. Davis, S. M. Nowick, "An Introduction to Asynchronous Circuit Design". Technical Report UUCS-97-013, Computer Science Department, University of Utah, Sep. 1997.

[14] J. Sparso, S. Furber, "Principles of Asynchronous Circuit Design: A Systems Perspective". European Low-Power Initiative for Electronic System Design. Kluwer Academic Publishers, ISBN: 0-7923-7613-7, Jan 2002.

[15] S. Furber, P. Woods, "Four-phase Micropipeline Latch Control Circuits". IEE Transactions on VLSI Systems, 4(2): 247-253, June 1996.

[16] T. Verhoeff, "Delay-Insensitive Codes – An Overview". Eindhoven University of Technology, January 1987.

[17] Handshake Solutions, http://www.handshakesolutions.com .

[18] K. Fawaz, T. Arslan, and I. Lindsay, "Implementation of Highly Pipelined Datapaths on a Reconfigurable Asynchronous Substrate", 2009 NASA/ESA Conference on Adaptive Hardware and Systems, 2009.

[19] K. Fawaz, T. Arslan, and I. Lindsay, "Conditional Acknowledge Synchronisation in Asynchronous Interconnect

Switch Design”, 2009 NASA/ESA Conference on Adaptive Hardware and Systems, 2009.

[20] J. Rose and S. Brown, "Flexibility of interconnection structures for field-programmable gate arrays," IEEE J. Solid-State Circuits, vol. 26, no. 3, pp. 277-282, Mar. 1990.

[21] GNU, Boston, MA, "GNU C compiler," 2005 [Online]. Available: http://gcc.gnu.org/

[22] Ying Yi , Ioannis Nousias , Mark Milward , Sami Khawam , Tughrul Arslan , Iain Lindsay, System-level scheduling on instruction cell based reconfigurable systems, Proceedings of the conference on Design, automation and test in Europe: Proceedings, March 06-10, 2006, Munich, Germany.

[23] Vaughn Betz , Jonathan Rose, VPR: A new packing, placement and routing tool for FPGA research, Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications, p.213-222, September 01-03, 1997.

[24] ARM Ltd., Cambridge, U.K., "ARM7 thumb family datasheet," ARM DOI 0035-3/02.02, 2002.

[25] S. Agarwala et al., "A 600-MHz VLIW DSP," IEEE J. Solid-State Circuits , vol. 37, no. 11, pp. 1532-1544, Nov. 2002.

[26] X. Li, B. K. Gunturk, and L. Zhang, "Image demosaicing: A systematic survey," in *Proc. SPIE-IS&T Electronic Imaging, Visual Communications and Image Processing*, Jan. 2008.

[27] J. W. Cooley, J. W. Tukey, “An algorithm for the machine calculation of complex Fourier series,” Math Comput. 19, pp. 297-301, 1965.

[28] D.W. Trainor, J.P. Heron, R.F. Woods, Implementation of the 2D DCT using a Xilinx XC6264 FPGA, IEEE Workshop on Signal Processing System SiP97.

[29] A. Viterbi. A personal history of the viterbi algorithm. IEEE Signal Processing Magazine, 23:120--142, 2006.

[30] G. Martinez, "TI TMS320VC5501/02 power consumption summary," Appl. Rep. SPRAA48, 2004.