2012 15th Euromicro Conference on Digital System Design

reMORPH – A Runtime Reconfigurable Architecture

Kolin Paul, Chinmaya Dash and Mansureh Shahraki Moghaddam
Department of Computer Science and Engineering
Indian Institute of Technology Delhi, India

***Abstract*—Programmable hardware built on a regular architecture can partially alleviate the problem of increased defect densities associated with transistor scaling by dynamically wiring around the defects [1]. The fine granularity of FPGAs is however unsuitable for effectively exploiting runtime reconfiguration because of the high overheads involved. A coarse grain reconfigurable array with malleable communication links – reMORPH – is proposed in this paper. The compute tile uses DSP48E and BRAM embedded blocks in a Xilinx FPGA and has a very low footprint of about 200 slice LUTs. The semi-systolic near neighbour communication interconnect can be dynamically reconfigured for each "epoch" of computation. The "epoch" or phases of the application are obtained via profiling or static data flow analysis. Some of the links between the compute tiles are changed during the reconfiguration phase which drastically reduces the context switch overhead enabling high performance/area applications to be built on this fabric.**

***Keywords*-Partial reconfiguration, FPGAs, programmable hardware**

I. Introduction

The continuous scaling of feature sizes has led to massive integration densities which now is in the order of a *billion* transistors per cm^2. This massive silicon real estate has enabled designers to implement huge on-chip caches, on-chip codecs and accelerators and many core architectures. However, this has come at a cost. The decreased feature sizes have led to increased power densities. One way to mitigate the increased power density levels is to operate the chip at low frequencies which essentially implies a sub optimal performance. Another major negative impact is with respect to increased defect densities and lower yields. The increase in complexity of modern chips gives rise to *"irregular"* circuits which increase defect probabilities. Regularity is a successful design paradigm in VLSI which results in lower number of defects. In most cases, the first devices manufactured at a new technology node point are very regular circuits like memories. The regularity in VLSI chips requires that the device be made customizable post fabrication giving rise to programmable circuits ranging from PLAs/PALs to FPGAs.

Programmable hardware, often synonymous with FPGAs, has been around for the last three decades. However, FPGAs are difficult to program and often do not meet performance constraints of power, area and time. In fact they have really been used as surrogate ASICs. The inherent programmability of FPGAs has never really been exploited despite the fact that partial runtime reconfiguration has been commercially available for at least a decade. Modern complex chips as found in many embedded applications, are large and often incorporate many "large" cores with hundreds of global lines. They often present a single interface to the external world via a common bus. Implementing such designs on an FPGA becomes very difficult and often "routing kills the implementation". The fragmentation of market and short time to market makes extreme demands on engineering efficiency. Industry and academia have realized the potential of increasing the granularity of the such regular blocks in the device to effectively meet the requirements of performance and time to market.

Therefore the key to effective use the available silicon real estate is to exploit regularity with coarse grained programmable logic. Coarse grain reconfigurable architectures have been researched for last 20 years and have been principally used as accelerators for compute intensive streaming signal processing applications. The key attraction of CGRAs is their near ASIC/hardware like computational efficiency and software like engineering efficiency. More importantly, many applications can be temporally partitioned by exploiting temporal locality in the code apart from data. This temporal partitioning allows significant area advantages by allowing effective reuse of the CGRA via runtime programming to do temporally distinct tasks. The regularity available in CGRAs also implies a certain degree of post fab programmability which can be exploited to *wire around the defects*[1]. Furthermore, once such a fabric is available, compilers can do very efficient application mapping. This makes it also possible for improved system level performance predictions.

The granularity of reusable objects has kept pace with Hemani's prediction of increasing $100x$ every decade [2]. The level of integration of hard and soft IPs in FPGAs has similarly increased over the years along with the intrinsic fabric switching speeds. The amount of configurable embedded block memory has enabled building of memory intensive accelerators. Fast carry chains and hard multipliers have also been used along with gigabit transceivers for implementing high performance digital processing systems. The Virtex 5 and 6 series (as also Spartan) FPGAs from Xilinx have lots of ALU-like DSP48E IP which operates nominally at 600MHz.

These characteristics have been used to propose a low context switch overhead run time reconfigurable architecture –**reMORPH** –composed of coarse grained modules (CGRM) connected together using malleable communication links. The semi systolic near neighbour shared memory communication architecture has been built to exploit partial runtime reconfigurability present in Xilinx FPGAs to achieve significant performance/area advantage while at the same time working with very low reconfiguration overheads. The basic compute element in reMORPH has a 5 stage pipeline using the DSP48E as the ALU and has 512 addressable data registers along with 512 instruction registers. The micro-sequencer works on an explicit instruction format enabling the processor to be built with a very low foot print of about 200 slice LUTs. Any application (re-)uses as many tiles as required for meeting performance constraints — the code executing in the processors as well as the interconnection network between the tiles is changed at application runtime according to a statically determined schedule. The main contributions of this work are

- A highly optimized processor core using DSP48E which can be programmed very easily using "*instructions*" rather than HDL
- A coarse grain reconfigurable array consisting of the cores mentioned above
- A malleable interconnection network which can be reconfigured at runtime
- A runtime reconfigurable multiprocessor which exploits partial dynamic reconfiguration

The paper is organized as follows. In the next section we review prior work in this area. In Section III the Coarse Grain Reconfigurable Architecture is described in detail. The next section briefly describes the compute model used in deriving the architecture as well in programming the same. Section V details some implementation aspects of the architecture. The last section summarizes the contributions of this work and provides pointers to future extensions.

II. Review

Most programmable systems of the recent past have been based on programmable memory like SRAM or flash memory. The contents of this memory determine the functionality as well as the connectivity of the circuit being implemented in the fabric. The process of populating boolean values 1,0 into the memory locations is know as *reconfiguration*. In some cases, the reconfigurable fabric allows portions of the system to alter its functionality at runtime by loading a different *partial configuration* into that region. This is termed as *Partial runtime reconfiguration (RTR)*. Lysaght et. al. developed an early system for partial reconfiguration using CPLDs for exploiting RTR [3]. The authors developed a generic hardware interface to exploit RTR available in Atmel AT6005 class of FPGAs. The overhead of serial programming of single context FPGAs is very high and hence time multiplexed configuration data using additional storage was developed. An early fine grained dynamically reconfigurable fabric called GARP [4] was developed at Berkeley. Many multi context devices and architectures have been proposed in the last fifteen years. Dehon et al [5] introduced DPGA which stored 4 contexts simultaneously. Early attempts at using multi context include Dharma [6] and Morphosys [7] among many others. Obviously the disadvantage of using multi context is the increased area overhead. The overhead of programming fine grained reconfigurable hardware is very heavy and hence there has been many attempts at building coarse grained architectures [8]. Piperench [9] is an architecture based on ALUs and is very well suited for stream based computations. The RAW architecture on the other hand uses a complex 8 stage single issue MIPS processor in each of the 16 programmable tiles connected by on chip network [10]. A similar architecture which uses 64 programmable logic blocks called nano processors was developed as a multimedia coprocessor [11]. A complete programming environment called SCORE based on the dataflow model was developed for stream oriented computing which supported phased reconfiguration and dynamic rate dataflow [12]. Many recent architectures developed in academia include Morpheus [13], Montium/Chameleon system from University of Twente [14], Adres [15] from IMEC in Belgium, Imagine [16] from Univ. of Stanford and REDEFINE from Morphing Machines [17]. A comprehensive survey of academic CGRAs has been done recently by Max Baron [18] for largely commercial CGRAs. Configuration management is an important issue in dynamically reconfigurable systems. Compton et. al. have investigated many issues in depth including caching, relocation and fragmentation [19].

Applications where dynamic reconfiguration has a prominent role to play have been developed for Phased Array Radar Processing System [20] and adaptive image filtering [21] among others. Lockwood et. al. used the idea of dynamic hardware plug-ins to develop efficient network routers [22].

III. A Coarse Grained Programmable Fabric

The proposed architecture – **reMORPH** shown in Figure 1, is a mesh of CGRMs connected in a nearest neighbor configuration.

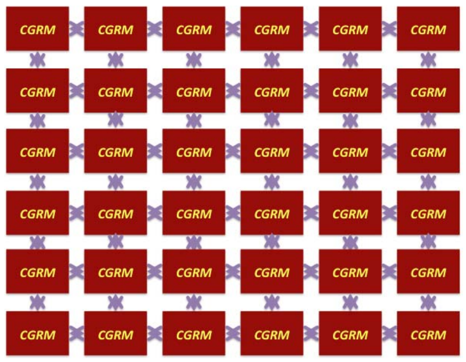

Figure 1. reMORPH

Clearly choosing the right granularity for the modules is of critical importance. All or some of the communication links can be reprogrammed at runtime. The choice of the appropriate processing element (CGRM) and the interconnection network plays an important role in reducing the reconfiguration overhead. We illustrate this with a small motivating example. Figure 2, illustrates a circuit which has a number of DSP48 blocks and BRAM blocks. The circuit topology evolves with time via only changes in the communication links – the colored(dotted) lines indicate the lines which change in each configuration. Table I shows the size of the difference bit stream when 1,2,3 and 4 links are changed in the example

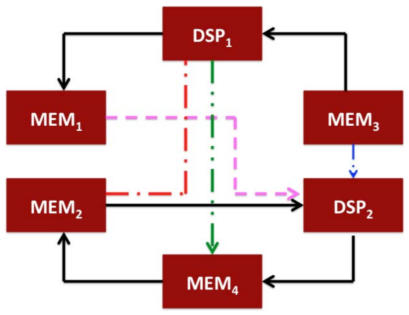

Figure 2. Example Circuit

circuit. The cost is also parameterized on the bus width. The size of the circuit is about 0.5 MB. Clearly if we only change the interconnection network during (partial) reconfiguration, the context switch overhead can be dramatically reduced.

TABLE I
COST OF RECONFIGURATION (BITS)

Link Width							
4	8	12	16	20	24	28	32
Changing One Link (Red)							
326	864	1094	1560	1608	2330	2888	3342
Changing Two Links (Red and Green)							
502	746	1176	1722	2172	2534	3038	3368
Changing Three Links(Red and Green and Blue)							
704	1188	1384	2050	2452	2980	3426	3388
Changing Four Links(all colored lines)							
696	1312	1738	2108	2422	3048	3472	3878

All applications go through "phases" defined by the communication patterns among the CGRMs in their lifetime. These phases, can for many applications (notably streaming ones) can be derived statically. Techniques to exploit runtime information to generate phase information have also been described in literature. The current architecture assumes that the application can be statically analyzed to generate all possible communication patterns. This temporal partitioning of the application is used to generate the configuration information necessary for each "epoch" of execution. The model of computation borrows from the ATM mode of communication where route discovery is done via packet switching and actual data transfer is done via circuit switched links. The architecture assumes that the routes are predetermined via an offline analysis method. This is a major advantage over an NoC implementation of communication as the architecture is capable of providing latency guarantees when compared to an NoC. The area overhead for implementing an NoC is also not encountered in this case. It may be noted that the change in interconnection network could also be accompanied by a change in the configuration data for each of the processing elements. The design of the processing element (CGRM) has also been done with the objective of ensuring that bit stream changes because of change in functionality is also minimized.

A. CGRM

The design of the processing element has been done keeping in view that the system would take advantage of the features available in modern FPGAs. The million plus logic cell chips (Figure 3) integrate a lot of memory which allows the building of small Harward style processors. The Xilinx chips also

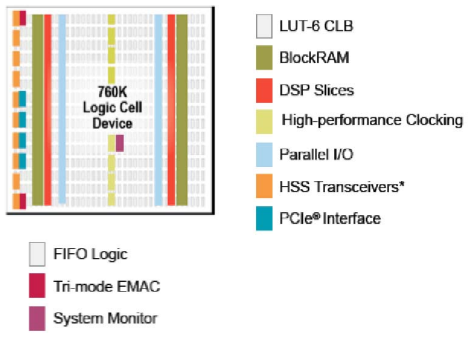

Figure 3. Xilinx FPGAs [23]

have a very high performance DSP48 element (Figure 4 which allows the building of a fast ALU. The DSP48E slice provides improved flexibility and utilization, improved efficiency of applications, reduced overall power consumption and increased maximum frequency. The high performance allows designers to implement multiple slower operations in a single DSP48E slice using time-multiplexing methods. The

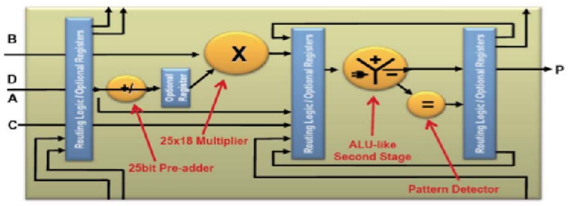

Figure 4. DSP48 [23]

DSP48E essentially has

- 25×18 2s complement multiplier
- 48 bit arithmetic and logic unit
- 48 bit comparator

The ALU-like control in the DSP48E allows for the implementation of many independent functions which include multiply accumulate, three-input add, barrel shift, wide-bus multiplexing, magnitude comparator, bit-wise logic functions, pattern detect, and wide counter. The architecture also supports cascading multiple DSP48E slices to form wide math functions, DSP filters, and complex arithmetic without the use of general FPGA fabric. The block RAM in Virtex-5 and 6 devices can be split into two 18K block RAMs. Each DSP48E slice aligns horizontally with an 18K block RAM. The opmode control bits of the DSP48E are used to implement the opcode of the instructions of this compute element. These two crucial elements – a fast ALU and lots of BRAM – are the key enablers to build the CGRM which can operate at frequencies in the range of 400MHz with a very low footprint. Each CGRM or grain is connected to its neighbour in one of the four principal directions at any instant in time. The links can change over the application's lifetime. As shown in Figure 5, the connectivity of the grains changes from a vertical down to vertical up connectivity over time.

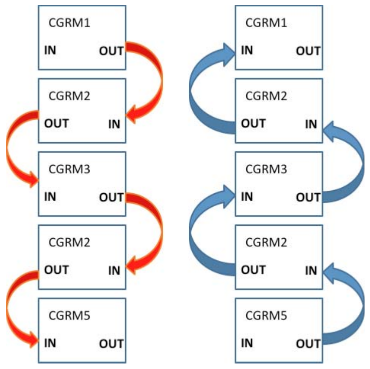

Figure 5. Runtime Configuration Change in Links

The coarse grain reconfigurable module is shown in Figure 6. Each tile reads data from its local memory but can

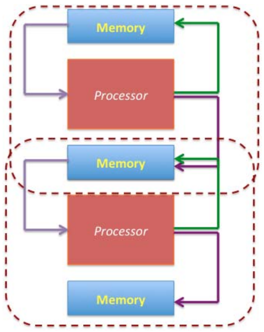

Figure 6. reMORPH Tiles

write to either its own memory or the neighbour's memory. This ability to write to neighbour's memory is similar to what systolic arrays traditionally do. The data generated at non neighbour grains is brought to the grain's memory using explicit copy instructions and changing connectivity if required. All the grains can in principle execute different instructions at every clock cycle which gives it a MIMD flavor. The processing element has explicit memories for instruction and data. A total of 512 instructions can be used to program the grain for an epoch. Each instruction can read data from 512 locations present in its Data memory.

B. MicroArchitecture of reMORPH tile

The internals of the grain are shown in Figure 7. This "Compute Element" based on the DSP48E works on 3 operands A (30 bits), B (18bits) and C (48 bits). The operation to be performed is decided by a 14 bit control word to the DSP48E1 block. The first positive clock edge to Compute Element registers all the inputs; the output P is calculated through the embedded functional blocks and made available at next clock edge. Effectively there is a two-stage execution unit. The number of stages for execution is kept at two for all classes of instruction. The comparator does not take part in generation of P, but it generates *"FLAGS"* for the control path. Those inputs of DSP48E1 which are not controlled by bits of control word are connected to '1' or '0' depending on whether they are active-high or active-low. It is the micro-sequencer logic which puts the control word and the operands at Compute Engines input-ports every clock cycle. Parallel use of two read ports of Block RAM enables a 72 bit instruction each cycle.

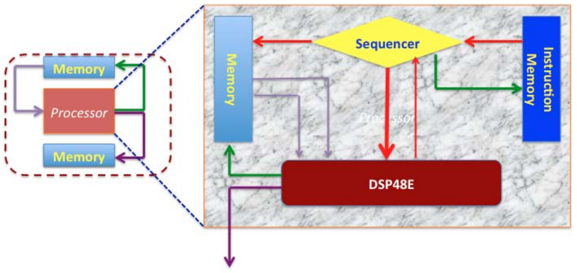

Figure 7. reMORPH Tile Details

The basic compute element or grain uses the instruction format shown in Table II.

All instructions of reMORPH can be seen as a combination of following fields

1) The Opcode [14 bits] which denotes the operation to be performed
   - Arithmetic
   - Logical
   - NOP
2) The Source Operands and Addressing Mode [20 bits]. Both direct and indirect addressing mode are supported.
3) The Destination[21 bits]
   - Write result To own register file
   - Write result to next grains register or both
   - The addressing mode is write back is to the own register file
4) The address of next instruction [11 bits] along with an enable bit which can be any one of the following
   - Sequentially next
   - Jump unconditionally to given address
   - Which flag to be checked in case of conditional jump
   - The HALT instruction

It may be noted that all the combinations of above four options (i.e. fields of instruction) are supported in the architecture. Some example instructions and the associated coding (omitting the all zero fields) is shown in Table III.

TABLE II
INSTRUCTION FORMAT

Control			Read Addr		Read Control			Write Control			Write Addr	Jump	
Opcode	ALU	CinSel	C	AB	Mode		A	WB	Addr Mode	Shared		JFlag	JAddr
7	4	3	9	9	1	1	1	1	1	1	9+9	1	9

TABLE III
INSTRUCTION EXAMPLE

Instn	Control			Read Addr		Read Control			Write Control			Write Addr	Jump	
	Opcode	ALU	CinSel	C	AB	Mode		A	WB	Addr Mode	Shared		JFlag	JAddr
r2=r1+r0	0110011000000000000000000000000011000100000000010000000000000													
	0xA3	0x0	0x0	0x0	0x1	1	0	0	0	1	0	0x2	0	0x0

The micro-architecture does not use any decoders in the data path and hence the instruction format explicitly specifies all the read and write addresses and modes of operation. For example, the write address is explicitly specified in two places (using 18 bits) – one is used when the processor writes to its own data memory (WB mode) and the other when it writes to that of the neighbouring compute element (shared mode). This redundancy was done to optimize the routing and mux costs which helped in generating smaller partial bit streams. This is also very useful in the context of using hard macro to define the CGRM. There are 4 bits which are free for future extensions. The Program Counter is incremented by '2' in each cycle for non-jump instructions; also the jump address must be an even number. When an instruction is issued from Instruction Memory (IM), the address of operand is put to data memory. Considering indirect addressing mode, the output of data memory is further used to supply the final operands to Compute Element. Thus operands are two clock cycles delayed and hence opcode(i.e. the control word) must also have two-stage registers in its path. The second stage of operand fetching is not used in case of direct addressing mode. However this stage is present and only does the registering of previous stage's outputs. When an instruction is issued from Instruction Memory, the address of operand is put to data memory. Clearly, the signals WriteBack, Sharing and Conditional JUMP wait for 4 cycles after issuing control word from Instruction Memory (so as to accommodate two stages of operand delay and two stages of execution). Thus there is a delay slot of 4 instructions to be handled by compiler. Two codes from the 8 possible jump settings (with the 3 bits) are reserved for 'no jump and 'unconditional jump. The remaining 6 codes are used for conditional jump depending on the flag Zero, Carry, Sign, Overflow and Underflow.

The micro sequencer controls the execution flow of the architecture. It is this block which controls the program counter, selecting the appropriate instructions and the operands to act on it. It also has a lot of other capabilities like where to write the output, unconditional and conditional jumps, selecting the appropriate addressing mode and control. As shown in Figure 8, the micro sequencer provides the control signals to the Instruction Memory, the Data Memory (register file) and the Program Counter.

The first version of reMORPH has been implemented on

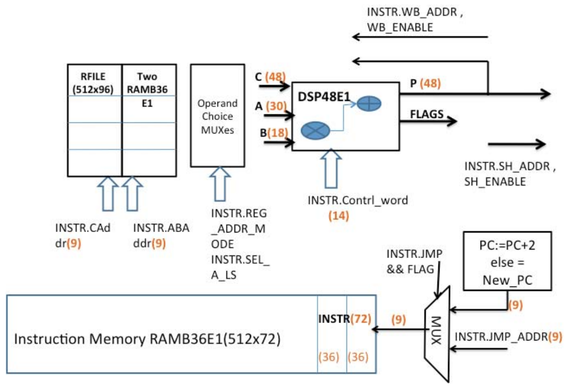

Figure 8. Sequencer

Spartan 6 based Digilent boards. The high level block diagram of the system is illustrated in Figure 9. All grains of the array

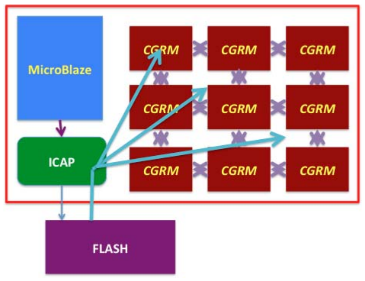

Figure 9. reMORPH System

must be programmed with instructions and initial data prior to any epoch of execution. The method and technology for filling memory must be as fast as possible so as to minimize reconfiguration cost. We access the ICAP interface by using the OPBHWICAP peripheral attached to the On-Chip Peripheral Bus and the operations of the ICAP are controlled by software running on a MicroBlaze on the FPGA.

In the simulation model of the current architecture, we provide at the top level for 72 bit data bus, separate write enables for data and instruction memory of each grain and 9 bit address bus connected to data memory. When top level write enables are active, they override grains internal signals. Instructions are stored sequentially using the PC to update address while writing. While programming data memory, the lower 48 bits of data in bus are written.

The architecture will suffer from the IO bottleneck when it is used to implement applications with large data sets. However, rapid advances in 3D integration can enable the building of a vertically stacked memory plane with vias directly connecting to each of the grains memory as shown in Figure 10.

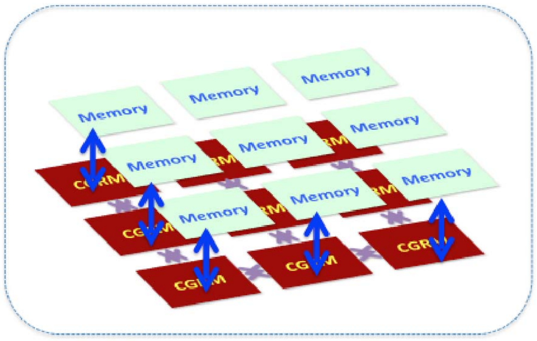

Figure 10. A 3D version of reMORPH

The proposed architecture – **reMORPH** is targeted to exploit runtime reconfiguration available in commercial FPGAs to enable building of high performance/area architectures. The reconfiguration is achieved either by

- Changing Instructions in Memory
- Changing connectivity between grains using a "**fast Programmable Interconnect**"

In general, the ability to reprogram the connectivity between "cores" in a fast and efficient manner is a core research issue in many core architectures and reMORPH illustrates how guaranteed latencies could be achieved in such architectures. In the next section, we describe the programming model used to develop small applications for this architecture.

IV. Compute Model

We model the application as a set of interacting sequential processes $\{p\} = \{p_1, p_2, \ldots p_k\}$. As has been mentioned above, the set of processes $\{p\}$ changes the pattern of interaction with each other over time. The application's communication patterns can be analyzed at compile time and phases of the application which have a common communication pattern can be identified either by static data flow analysis or by profiling. This set of processes $\{p\}$ is mapped onto the set of compute elements (grains) $\{P\} = \{P_1, P_2, \ldots P_k\}$ for each phase or epoch. The given application needs to be placed and routed on the available grains on the device fabric – this is achieved by configuring the programmable elements in the device which is called the "*configuration data*". Let C_i denote a configuration of k compute elements. The configuration is composed of the BRAM contents (sequence of micro instructions) of each of the P_k elements as well as the configuration data for the interconnects. The coarse grain reconfigurable architecture assumes a "fast" method of reconfiguring the "switches" responsible for the changing the interconnection network between the different compute elements. Figure 11 illustrates the fact that the same set of processes $\{p\}$ are mapped to the fabric in configuration C_i for the i^{th} epoch and then subsequently mapped to the configuration C_j for the j^{th} epoch and so on.

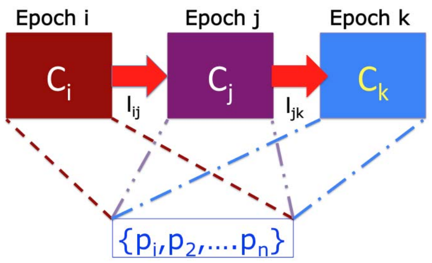

Figure 11. Epochs

The granularity of the grains allows for the execution of a large number of instructions before a configuration change is mandated. This change in configuration could happen because of

- A new piece of code being scheduled on the grain which causes a change only in the BRAM contents
- A change in the communication pattern among the different compute elements

Configuration data can in modern FPGAs be loaded in parallel — nevertheless reducing the size of the partial bitstream is of immense importance.

We now consider there is no change in code executing in each of the processes to analyze the cost of adapting to a new communication pattern. A configuration C_i remains active for time period τ_i^a before a configuration change happens as shown in Figure 12. A configuration change C_i to C_j incurs

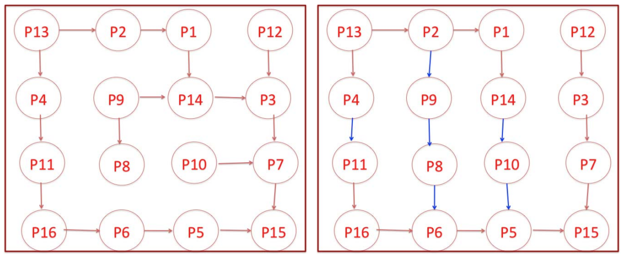

Figure 12. Configuration C_i and C_j

a cost τ_{ij} which is proportional to the change in the number of communication links l_{ij}.

The runtime of the application is given by

$$Runtime = \underbrace{\sum_{C_i} \tau_i^a}_{A} + \underbrace{\sum_{C_i, C_j} \tau_{ij}}_{B} \quad (1)$$

The first term is the sum of all the run times in each epoch while the second term reflects the reconfiguration cost of all the context switches. Under the assumption that all the $\tau_i s$ are known the term A in Equation 1 is statically known. The term B is dependent on the interconnection network between the grains as also the programs executing on the grains. Therefore careful placement of the p_i's to the P_k compute elements can help in reducing the overall runtime. We now outline an algorithm that minimizes the configuration time between the epochs. Each of the C_i's corresponds to the communication amongst the different processes in an epoch. The p_i's need to be mapped to the P_k's (which are the compute elements) in a manner such that the change from C_i to C_j is minimized. A simulated annealing algorithm is used to determine the mapping $p_i \longrightarrow P_k$ such that $\sum_{\forall(ij)} \tau_{ij}^c$ is minimized, the details of which are omitted for lack of space.

In general, different assignments of $p_i \longrightarrow P_k$ for a C_i will result in different τ_i^a which would affect the overall runtime. This is because data produced by process p_i in processor P_k ($i \neq k$) in epoch i will have to be copied to processor P_j for process p_i to execute in the next epoch.

In the next section, we describe the implementation of reMORPH on a Xilinx FPGA.

V. Implementation Details

The grains are carefully placed and routed using placement constraints so that they can operate at the peak frequency. Figure 13 shows the physical placement of one of the grains where the positions of DSP48E and BRAMs have been specified. Figure 14 shows 3 grains compactly placed

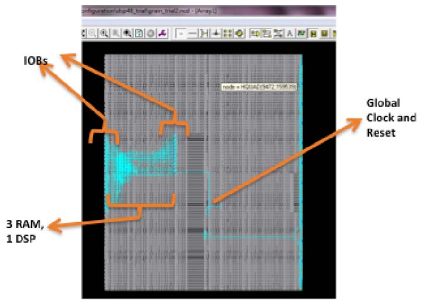

Figure 13. One Grain

in the fabric. The usage of BRAM and other associated resources is shown in Table IV. Clearly, the blocking resource

TABLE IV
Resource Summary

Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	41	93,120	0%
Number of Slice LUTs	196	46,560	0%
Number of RAMB36E1/FIFO36E1s	3	156	1%
Number of bonded IOBs	167	360	46%
Number of DSP48E1s	1	288	1%

appears to be the amount of block RAM available. About 40 tiles can fit in the relatively small Xilinx Spartan 6 LX45 FPGA. It may be mentioned here that while we reconfigure links, the partial bit stream is composed only of the changes

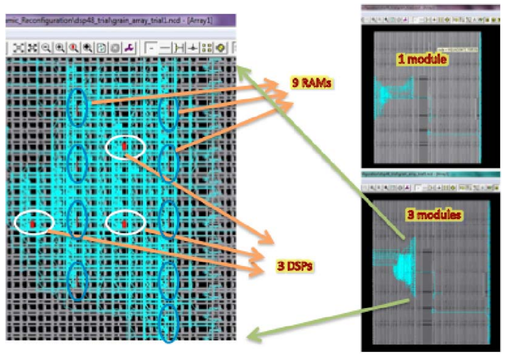

Figure 14. Three Grains

where the links change. The width of these links is about 20 for each link. As shown in Table I, the cost incurred in very small. Practically the achievable data transfer rates using the ICAP interface is about 180MB/s which ensures that the reconfiguration overhead is of the order of tens to hundreds of cycles only. Many small applications have been implemented in this architecture. Since complete C-style loops are supported, many (parallelizable) compute kernels have been ported including summation of N numbers, factorial, matrix multiplication, 1D FFT etc. The following subsection illustrates a method of implementing a map reduce type of application on reMORPH.

A. Sum

The addition of 1000 numbers using 4 CGRA tiles can be done as shown in Figure 15. There are three ways to perform

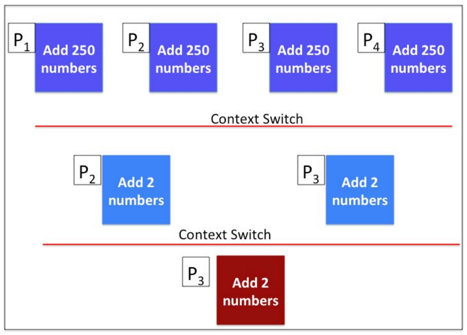

Figure 15. Addition Example

the addition on reMORPH:

- The grains/tiles and interconnection are configured once and no change happens in the runtime of the application
- The interconnection network and the code in each tile changes during different epochs
- The interconnection network is only reconfigured during context switches in different epochs

As can be seen in Figure 15, when there is a context switch, different code could execute in the following epoch. This code can be predicated with the clock cycle in which case

TABLE V
EXAMPLE: SUM OF A 1000 NUMBERS

Tasks \ Configuration	One	Two	Three
	Cycles		
Initialization	110	110	110
Code Download	39	28+3+8	36+42
Data Download	4×264=1056	1056	1056
Reconfig Overhead	0	10	10
Program runtime	1910	8	22
Total Time	3115	3120	3155

no reconfiguration of the instruction memory is mandated. Further, it may be observed that in general, different processors could execute different code in different epochs. Hence careful assignment of processes to processors reduces the data movement/reconfiguration costs. In this case, for example, in the second epoch, the code executes in processors 2 and 3 because in the previous epoch, the processes p_1 and p_2 wrote the sums to the register file/data memory of processor P2 whereas p_3 and p_4 wrote to that of processor P3. In the third epoch, the need for reconfiguration/context switch is trivially decided.

In this case, the number of cycles needed to compute the sum in the three cases is given in Table V.

VI. CONCLUSION

Partial runtime reconfiguration has not been exploited for building high performance applications primarily because of the reconfiguration overhead. This overhead of the context switch across temporal partitions of the application can be reduced with coarse grain reconfigurable architectures. In this paper, we presented a CGRA built using advanced features of Virtex FPGAs and which effectively exploited partial dynamic reconfiguration. A small map-reduce type application was described to demonstrate the working of this architecture. In the future, a memory hierarchy will be built using 3D integration to enable processors to work with large data sets. A process calculus framework is being developed to describe applications which can be mapped onto to this architecture "automatically".

ACKNOWLEDGMENT

The work has been done under the CREST project funded by DST-VINNOVA. The authors thank the anonymous reviewers and M Balakrishnan and Anshul Kumar for valuable feedback.

REFERENCES

[1] R. Jain, A. Mukherjee, and K. Paul, "Defect-aware design paradigm for reconfigurable architectures," in *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*, vol. 00, p. 6 pp., march 2006.

[2] A. Hemani, "Charting the eda roadmap," *IEEE Circuits and Devices*, vol. 20, pp. 5–10, 2004. QC 20120202.

[3] Patrick Lysaght, Hugh Dick, Gordon McGregor, David McConnel, and Jon Stockwood , "Prototyping Environment for Dynamically Reconfigurable Logic.," in *Field Progammable Logic* , 1995.

[4] T. J. Callahan, John Hauser, and John Wawrzynek, "The Garp Architecture and C Compiler," *IEEE Trans. on Computers*, April 2000.

[5] E. Tau, D. Chen, I. Eslick, J. Brown, and A. DeHon, "A first generation dpga implementation," in *In Proceedings of the Third Canadian Workshop on Field-Programmable Devices*, pp. 138–143, 1995.

[6] N. Bhat, K. Chaudhary, and E. S. Kuh, "Performance-oriented fully routable dynamic architecture for a field programmable logic device," Tech. Rep. UCB/ERL M93/42, EECS Department, University of California, Berkeley, 1993.

[7] H. Singh, M. hau Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, pp. 465–481, 2000.

[8] J. Becker and R. Hartenstein, "Configware and morphware going mainstream," *J. Syst. Archit.*, vol. 49, no. 4-6, pp. 127–142, 2003.

[9] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. R. Taylor, "Piperench: A virtualized programmable datapath in 0.18 micron technology," in *In Proc. Of IEEE Custom Integrated Circuits Conference*, pp. 63–66, 2002.

[10] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The raw microprocessor: A computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, pp. 25–35, Mar. 2002.

[11] T. Miyamori and K. Olukotun, "Remarc: Reconfigurable multimedia array coprocessor," in *IEICE Transactions on Information and Systems E82-D*, pp. 389–397, 1998.

[12] E. Caspi, M. Chu, Y. Huang, J. Yeh, Y. Markovskiy, A. Dehon, and J. Wawrzynek, "Stream computations organized for reconfigurable execution (score): Introduction and tutorial," in *in Proceedings of the International Conference on Field-Programmable Logic and Applications*, pp. 605–614, Springer-Verlag, 2000.

[13] F. Thoma, M. Kuhnle, P. Bonnot, E. Panainte, K. Bertels, S. Goller, A. Schneider, S. Guyetant, E. Schuler, K. Muller-Glaser, and J. Becker, "Morpheus: Heterogeneous reconfigurable computing," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pp. 409 –414, aug. 2007.

[14] G. J. M. Smit, M. Bos, P. J. M. Havinga, S. J. Mullender, and J. Smit, "Chameleon – reconfigurability in hand-held multimedia computers," in *Proc. First International Symposium on Handheld and Ubiquitous Computing, HUC'99*, 1999.

[15] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix.," in *FPL*, vol. 2778 of *Lecture Notes in Computer Science*, pp. 61–70, Springer, 2003.

[16] J. Owens, S. Rixner, U. Kapasi, P. Mattson, B. Towles, B. Serebrin, and W. Dally, "Media processing applications on the imagine stream processor," in *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*, pp. 295 – 302, 2002.

[17] A. Satrawala, K. Varadarajan, M. Lie, S. Nandy, and R. Narayan, "Redefine: Architecture of a soc fabric for runtime composition of computation structures," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pp. 558 –561, aug. 2007.

[18] M. Baron, "Trends in the use of re-configurable platforms," *Design Automation Conference*, vol. 0, pp. 415–415, 2004.

[19] Z. Li, K. Compton, and S. Hauck, "Configuration caching management techniques for reconfigurable computing," in *In IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 22–36, 2000.

[20] E. Seguin, R. Tessier, E. Knapp, and R. W. Jackson, "A dynamically-reconfigurable phased array radar processing system," in *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications*, FPL '11, (Washington, DC, USA), pp. 258–263, IEEE Computer Society, 2011.

[21] Nitin Srivastava, Jerry L. Trahan, Ramachandran Vaidyanathan and Suresh Rai, "Adaptive Image Filtering using Run-Time Reconfiguration," in *Proceedings of RAW'03*, 2003.

[22] E. L. Horta, J. W. Lockwood, D. E. Taylor, and D. Parlour, "Dynamic hardware plugins in an fpga with partial run-time reconfiguration," in *Proceedings of the 39th annual Design Automation Conference*, DAC '02, (New York, NY, USA), pp. 343–348, ACM, 2002.

[23] "Xilinx Inc ," Information available at http://www.xilinx.com/.