2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum

Design and Implementation of High Performance Architectures with Partially Reconfigurable CGRAs

Mansureh Shahraki Moghaddam, Kolin Paul and M Balakrishnan
Department of Computer Science and Engineering, IIT Delhi, India

ABSTRACT

Programmable hardware built on a regular architecture can be used to address the challenges associated with using many fixed core architectures for applications which have varying compute power requirements during the lifetime of execution. The fine granularity of FPGAs is however unsuitable for effectively exploiting runtime reconfiguration because of the high overheads involved. Effective use of a dynamically reconfigurable fabric across a range of applications remains a challenge. In this work, we use a model coarse grain reconfigurable fabric to explore the potential of such a fabric for a range of key reconfiguration parameters. This coarse grain reconfigurable array with malleable communication links is used for design space exploration of two compute intensive kernel implementations which exploit dynamically reconfiguration. The semi-systolic near neighbour communication interconnect can be dynamically reconfigured for each "epoch" of computation. Different blocks of the application program reuse the compute grain in different epochs. Some of the links between the compute tiles are changed during the reconfiguration phase and because the architecture is partially reconfigured, the reconfiguration in some tiles can be completely overlapped with computation in other tiles. The paper proposes a methodology to exploit this design paradigm to drastically reduce the context switch overhead for rebalancing the pipeline to build high performance/area applications on this fabric.

Keywords

Dynamic Reconfiguration,CGRA, FFT, JPEG encoder

1. INTRODUCTION

The continuous scaling of feature sizes has led to massive integration densities which has resulted in many "IP's" being present in the same die/chip. To take advantage of the large number of such ip cores, *programmability*, available post device fabrication is of crucial importance. Apart from the ability to potentially improve yield in the sub $22nm$ era, the ability to make the device processor architecture customizable post fabrication can also help improve the performance/watt figure of merit.

Programmable hardware, often synonymous with FPGAs, has been around for the last three decades. FPGAs are however, too fine grained in nature and hence are difficult to program. Because of limited routing capabilities and the fine grain nature of the FPGA fabric, it often becomes difficult to meet performance constraints of power, area and time. It is not a coincidence that FPGAs are primarily looked upon by the industry for implementing low volume "ASICs" that are not too power sensitive.

Effective use of the available silicon real estate at current technology node points and below would require the designer to exploit regularity. The fabric should be based on coarse grained programmable logic. Coarse grain reconfigurable architectures(CGRA) have been well studied in literature principally in the context of accelerators for compute intensive streaming signal processing applications because of 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. An application can be represented as a set of sequential communicating processes which can then be placed/mapped onto an array of coarse grain compute tiles. Depending on the required performance (throughput), each process can be (trivially) mapped onto a tile. If however, area (and power) is a constraint, then tiles may be reused to temporally pipeline the processes. Significantly, using active partial reconfiguration allows the dynamic balancing of the compute pipeline on the basis of ambient conditions.

The major contribution of this paper is to advance understanding of the potential of dynamic reconfiguration in the context of CGRAs. This has been studied by implementing two well documented compute intensive applications and analysing the performance metrics under various area and reconfiguration cost constraints. Recently, a new CGRA called **reMORPH** [1] has been proposed where the authors have effectively used the dynamic reconfiguration capabilities of Xilinx FPGAs to build a high performance coarse grain fabric. In the next section, we briefly review reMORPH which is an architecture that is dynamically reconfigurable and can instantiate compute elements on demand to meet performance/area requirements to set the context for this paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

978-0-7695-4979-8/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPSW.2013.121

IEEE computer society

2. A COARSE GRAINED PROGRAMMABLE FABRIC

A typical reconfigurable multiprocessor framework consists of compute tiles with a programmable interconnect.

It has been reported that if the reconfiguration is limited to changing the connectivity at runtime, the overheads are typically very low [1]. This allows fairly large circuits to be implemented modularly in a time multiplexed manner which makes the implementation area and power efficient. This is because most applications go through "phases" defined by the communication patterns among the Coarse Grain Reconfigurable Modules (CGRM) in their lifetime. These phases, for many applications (notably streaming ones), can be derived statically. The connectivity between the tiles changes in each of the phases and can be statically derived. This temporal partitioning of the application is used to generate the configuration information necessary for each "epoch" of execution. It may be noted that the change in the interconnection network could also be accompanied by a change in the configuration data for each of the processing elements.

The partially reconfigurable nearest neighbor mesh connected array of coarse grain reconfigurable tiles **reMORPH** [1] is illustrated in Figure 1. Modern FPGAs have hard DSP macros and lots of embedded memory which have been used to design the processing element(PE)/tile/grain of this architecture to operate at 400 MHz with a very low footprint of 200 slice LUTs

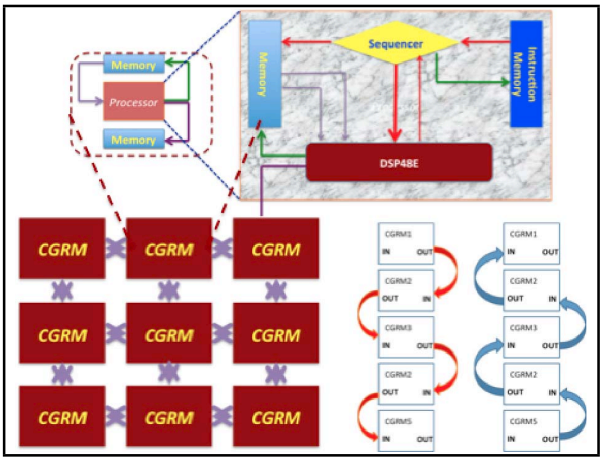

Figure 1: reMORPH [1]

Each tile 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. Each tile reads data from its local memory but can write to either its own memory or the neighbour's memory. The data generated at non neighbour tiles is brought to the tile'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. Two 512×48 sized dual port block RAMs with two parallel reads and one write capability is used to store the data (data memory). One 512×72 sized dual port BRAM is used as the instruction register. Semi-systolic near-neighbour shared memory communication involves least amount of resources used for routing enabling the array to be clocked at 300-400 MHz (depending on the speed grade of the device). Each PE can implement arithmetic and logic operations along with direct and indirect addressing. This enables complete 'C' style loops to be executed by the PE which supports these operations on a 48 bit word. Memory locations are reused to store the intermediate results. In each iteration, the same set of instructions are executed by updating the base addresses of the registers to read new data using register indirect addressing. All the grains can in principle execute different instructions at every clock cycle which gives it a MIMD flavor.

This streaming multi-processor architecture 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**"

The high level block diagram of the prototype system is illustrated in Figure 2. The application code implemented

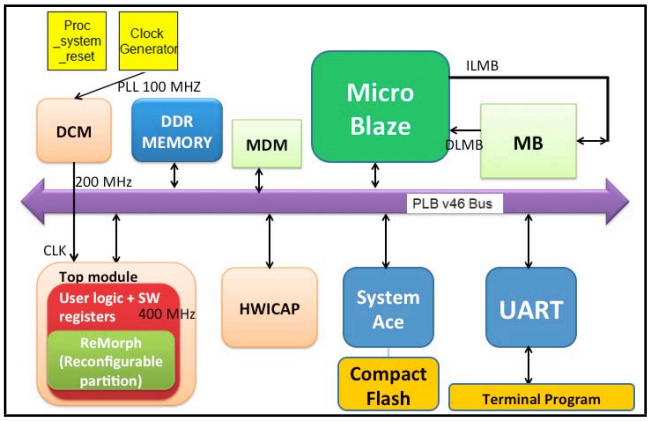

Figure 2: System Diagram

as an accelerator is mapped to the reconfigurable partition which is attached as a peripheral attached to the PLB bus. The bus master is a soft processor — MicroBlaze — which acts at the overall Runtime Management System Controller and helps in performing IO with the external world via the UART and other available interfaces. The ICAP interface is used to load the partial bit streams from the Compact Flash (using the Systems ACE controller) necessary to reconfigure the connectivity at runtime according to the algorithm running on MicroBlaze and can achieve data transfer rates in excess of 180MB/s [2].

We model the application as a set of interacting sequential processes $\{p\} = \{p_1, p_2, \ldots, p_k\}$. The pattern of interaction among the processes changes 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.

A configuration C_i remains active for time period τ_i^a before a configuration change happens. A configuration change C_i to C_j incurs 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

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

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. The third term C takes care of copying data across tiles in the case where the producer and consumer tiles are not neighbours. Therefore careful placement of the $p_i's$ to the P_k compute elements can help in reducing the overall runtime. 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. 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. This is taken care of in term C of Equation 1.

We measure the effectiveness of mapping algorithms and the dynamically reconfigurable CGRA architecture by implementing two well documented application kernels. Algorithms from the image processing domain can be effectively modelled using a process network framework. For example, the JPEG encoder can be described as a combination of the processes { *Blocking/shift, DCT, Quantization, Zig Zag, Huffman*} operating in a pipelined manner as shown in Figure 3.

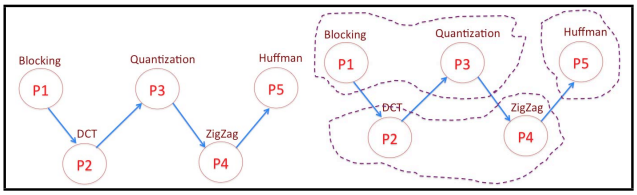

Figure 3: JPEG Encoder

Each of these blocks can be implemented in one reMORPH tile. However, the architecture allows for the rebalancing of the spatial pipeline using a temporal one by (re-)using a tile for performing different blocks at different times. One such grouping using 3 tiles is shown on the right side of Figure 3.

Similarly, in the signal processing domain, transform kernels are the the key computational kernels. For example, the FFT kernel is used in a myriad of applications and can be described as a collection of butterfly processes (BF) as shown in Figure 4.

Different mappings of the processes to the tiles gives us different performance metrics.

In the next section, we describe the design space that can be explored to obtain an optimal implementation on such an architecture.

3. DESIGN SPACE EXPLORATION

The mapping of processes to tiles is a hard problem when we consider that the connectivity between the tiles can be changed at runtime. The design space that needs to be looked at increases substantially. We look at two applications — FFT which is data and compute intensive and JPEG which is computationally very costly.

3.1 Radix 2 FFT Decomposition

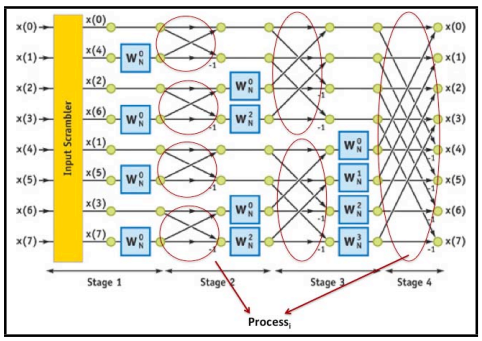

Figure 4: FFT Processes

Radix2 FFT has a flat two-dimensional computational structure, which makes it easy to break the structure horizontally or vertically to smaller processes which are the same but work on different parts of overal computation. The 16-point Radix2 FFT example depicted in Figure 5 shows that the structure of FFT lends itself easily to a pipelined implementation on an array of processing tiles. The processing element in reMORPH has 512 words for instruction and data separately. which has to be used for complex inputs and for storing twiddle factors. The outputs typically reuse the input memory locations.

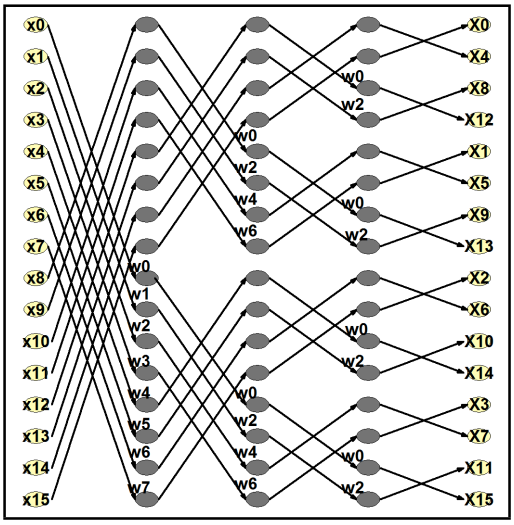

Figure 5: 16-point R2FF

A tile can in general, do M complex operations on M complex inputs; we will later compute exact amount of M for our specific tile. Therefore the N point Radix2 FFT structure has to be broken into $\frac{N}{M}$ horizontal partitions (rows), each mapped to atleast one tile. Also for an N-point Radix2 FFT there are log_2^N stages which the inputs go through. These two parameters, M and N, are key parameters in deciding number of tiles used in a circuit and task allotted to each one of them.

We do a design space exploration for a 16 point Radix2 FFT to identify the key issues that need to be taken care of in using a CGRA for implementing the kernel. Rearranging the inputs of 16-point Radix2 FFT of Figure 5 will give the partitioned structure of Figure 6. In this figure M and N are

4 and 16 respectively. So the structure is broken into four rows or partitions, each with four stages. Depending on the performance-area criteria specified by designer, atleast one and atmost four tiles can be used to execute operations allotted to these four stages. For this case, the partition size is four, which indicates that the partition input length is four complex values.

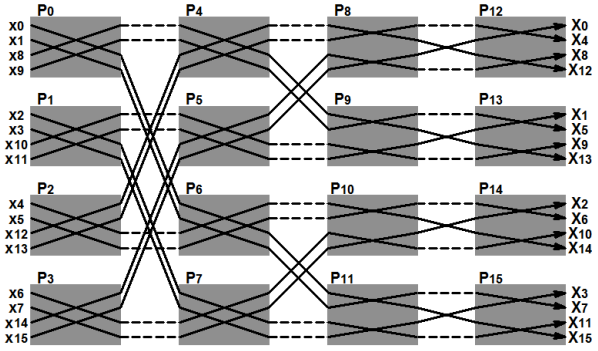

Figure 6: 16-point R2FFT partitions

Figure 7 proposes four different mappings of the above example into tiles of partition size $M = 4$. We use rectangles to denote tile and circles in them indicate stage. The four cases show

a) four tiles are used each doing all four stages of a row

b) 16 tiles are used; one tile for each stage in each row

c) 8 tiles are used in 4 rows and 2 columns with equal distribution of stages in each row

d) 8 tiles are used in 4 rows and 2 columns with unequal distribution of stages in each row

Clearly the decomposition proposed in the last case (d)is not good for pipelined implementation. Mapping each partition to a tile in all models except the one in bottom-right corner, are good candidates for a pipelined implementation, because the complexity of R2FFT structure is decomposed into partitions uniformly. The decision of selecting one of the possible mappings depends the optimization objective which can either be area or throughput. Clearly, Figure 7, indicates that while higher performance is expected if the number of tiles is increased, arbitrarily increasing the number of tiles can lead to a greater number of links which must be reconfigured on-the-fly. Therefore, the final performance metric becomes a function of both the number of tiles and link reconfiguration cost.

In general, to implement an N-point Radix2 FFT using tiles of size M, computational structure of FFT has to be broken into $\frac{N}{M}$ rows. In each row there is $log_2 N$ stages each doing M complex operations on M complex inputs to produce M complex outputs. Irrespective of the number of columns, tiles in each column transfer the computed complex outputs to the next column. When N is greater than M, before transferring results to next column, the generated outputs must be reordered. It may be noted that each tile transfers only half of its output to another tile. While process BF_i does the computation part of stage i, *vcp* and *hcp* processes are used to exchange data vertically with another tile in the same column or send data horizontally to next tile of the right-side column. For example, in the 1024-point

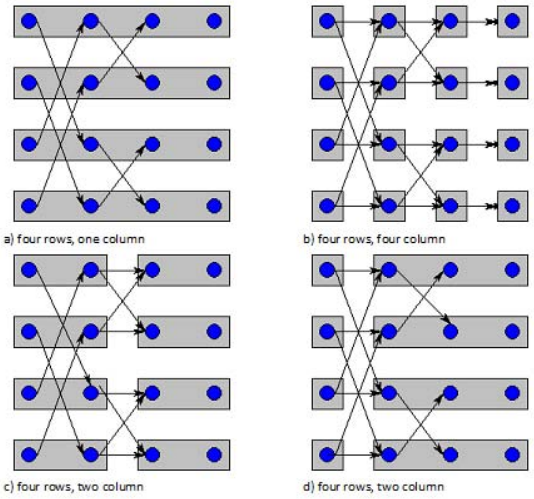

Figure 7: Different Mappings for16-point Radix2 FFT

Radix2 FFT, the runtime of the butterflies executing in the processing elements or tiles in different stages is not exactly the same (Table 1), even though the same number of complex operations are executed in them. The main reason for this minor difference is different structures used in our implementation for the M-point butterfly in different stages.

Table 1: 1024-point Radix2 FFT processes

process	runtime(ns)	Twiddle	insts	dmem
BF0	2672	128	101	128*2+41 +twiddles
BF1	2672	128		
BF2	2672	128		
BF3	4112	64		
BF4	3434	32		
BF5	3134	16		
BF6	3062	8		
BF7	3182	4		
BF8	3554	2		
BF9	4364	1		
vcp	789	0	16	11
hcp	1557	0		

For an N-point R2FFT using tiles of size M, we need at least one and at most $log_2 N$ columns, each of size $\frac{N}{M}$ tiles. BF_i process needs 2M locations for M inputs and at most M locations for $\frac{M}{2}$ twiddle factors and 41 locations for temporary locations. Depending on whether we reuse the input data locations to store the outputs or not, we need $3 \times M + 41$ or $5 \times M + 41$ locations in data memory of each tile. Therefore $M = 2^x$ where $x = \lfloor \frac{DM-41}{3} \rfloor$for a tile with data memory of size DM. For the specific case of reMORPH where DM=512, M turns out to be 128. Therefore a 1024-point Radix2 FFT implementations needs atleast 8 and at most 80 tiles. Of course we could consider more than one copy of this structure to increase throughput while sacrificing area.

The code part for an M-point complex operation in each stage of a Radix2 FFT is small enough to be saved permanently in the instruction memory of a tile. Tiles in the first column receive their input in external (preprocessing) column and the tiles in other columns receive input from the

tiles in previous column. But $\frac{N}{2}$ twiddle factors which are complex numbers can not be completely loaded into the data memory of the tiles during preprocessing phase. Therefore they have to be loaded partially during runtime. Table 1 states the exact number of twiddle factors needed for each stage. Before each tile starts executing the next stage BF process, the required twiddle factors must be loaded into tiles' data memories. This overhead is reduced by effectively overlapping computation with communication. Figure 8 illustrates the twiddle factors used in each stage of each row for 64-point Radix2 FFT when M=8.

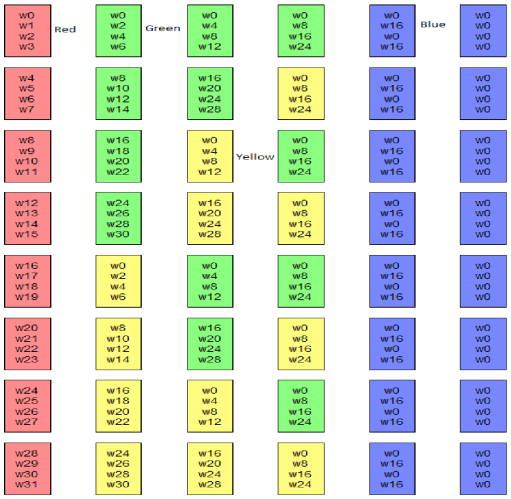

Figure 8: Butterfly for 16-point R2FFT

To reduce the overhead of loading twiddle factors in to the data memory before each stage's execution, we run the following algorithm:

- Twiddle factors for first column (Red ones), can be loaded in preprocessing phase
- Twiddle factors for next three column are of two types; Green and Yellow. As $w_{2i} = w_i^2$, a green tile during execution stage k can generates twiddle factors for stage k+1. Therefore while we need to reload data memory with yellow twiddle factors, green twiddles are generated automatically. According to runtime frequency of our circuit and the reconfiguration data transfer rates, while reloading one location in data memory takes 33.33 ns, executing an instruction takes 2.5 ns. Therefore it is great help in generating new twiddle factors from previous twiddle factors instead of loading memory with new twiddles
- Twiddle factors for last two column (Blue ones) are already in data memory, only index to go through them is changed from column to column

So instead of reloading $N \times log_2 N$ we just need to reload $(log_2 N - log_2 M) \times \frac{N}{2}$ twiddle factors (for yellow tiles), which is a considerable reduction in data memory loading cost.

The same reasoning (described for green tiles) is used in our implementation to reduce cost of reloading data memory for *copy* process variables. Copy process *vcp* will be executed many times during an application's lifetime, and each time we need new *source* and *destination* variables to copy data from a source tile to a destination tile. So instead of reloading data memory with new values for these two variables, it is more beneficial to update these two variables using current *vcp* process, Table 2.

Table 2: Optimized Copy Processes

cols	prev. cost(ns)	new cost(ns)	improvement
1	1066.6ns	15.0ns	1051.6ns
2	1066.6ns	15.0ns	1051.6ns
5	533.3ns	10.0ns	523.3ns
10	0.0ns	0.0ns	0.0ns

A link is a set of nets connecting two tiles in our circuit, and it is of size 48 lines for a word size of 48-bits. As illustrated in Figure 6, each partition may send its output to a different tile on to the column on its right. Tile p0 sends to p4 and p6, and p2 also sends to p6 and p4. Therefore two *link reconfigurations* are required in each column. To optimize link reconfiguration, we can first exchange half of the output among each pair of tiles in a column (Figure 9) and then each tile can transfer the final output to the tile in the next column. While in this case also we need two link reconfigurations, but the vertical link reconfiguration will overlap BF process execution time. Even if many columns are used in our circuit we do not need to do data exchange vertically for all columns. It is sufficient to do vertical link reconfiguration and vertical data exchange only for the first $log_2 N - log_2 M$ columns. Because after that exchange must happen inside a tile and instead of reordering data in a tile data memory, we can just adjust the BF_i process loop index in tile i.

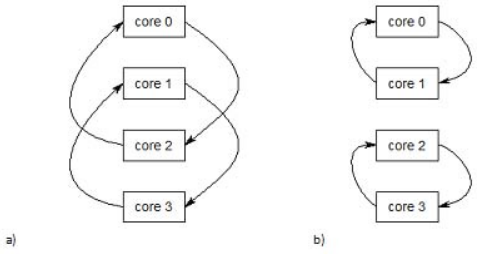

Figure 9: Vertical links in a column

3.2 Empirical Performance Equation

In an N-point Radix2 FFT implementation total execution time (τ) consists of time to receive data from input circuit (τ_0), time that the ButterFly BF (τ_1) process executes including reconfiguration times if required, vcp (τ_3) and hcp (τ_6) processes, time to stablish links (τ_2 and τ_5) and finally time to execute the mentioned processes (τ_2 and τ_4). Following equations give the cost of mentioned tasks.

$$\tau = \sum_{i=0}^{7} \tau_i \tag{2}$$

S_i which is used in calculation τ_2 is one only for $log_2^N - log_2^M$ first columns. t_{cp}^r and $t_{BF_i}^r$ are runtime for *vcp*, *hcp* and BF_i processes respectively, as stated in Table 3. reg_{cp} is number of copy process variables which need to be reloaded.

$$S_i = \begin{cases} 1 & , if\ i < 3 \\ 0 & , otherwise \end{cases} \tag{3}$$

t_l is time to configure links for a column whether inside the column or from the column to the next column. L is cost of reconfiguring of size 48 wires.

$$t_l = \frac{N}{M} \times L \tag{4}$$

t_d is time to reload data memories of one column tiles with new values for *cp* process *source* and *destination* variables.

$$t_d = reg_{cp} \times \frac{N}{M} \times 33.33 \tag{5}$$

τ_0 is runtime for *hcp* process.

$$\tau_0 = t^r_{cp} \tag{6}$$

τ_1 is time to reload twiddle factors to *yellow* tiles.

$$\tau_1 = \frac{N}{2} \times 33.33 \times \begin{cases} 3 & ,if cols = 1 \text{ or } 2 \\ 2 & ,if cols = 5 \\ 0 & ,if cols = 10 \end{cases} \tag{7}$$

$$A = \sum_{i=0}^{9} Max(t^r_{BF_i}, S_i \times t_l)$$

$$B = \sum_{i=0}^{4} Max(t^r_{BF_i}, t^r_{BF_{i+5}}, S_i \times t_l)$$

$$C = \sum_{i=0}^{1} Max(t^r_{BF_i}, t^r_{BF_{i+2}}, t^r_{BF_{i+4}}, t^r_{BF_{i+6}}, t^r_{BF_{i+8}}, S_i \times t_l \times (2-i))$$

$$D = Max^9_{i=0}(\left\{ t^r_{BF_i} | i = 0, 1, ..., 9 \right\}, 3 \times t_l)$$

τ_2 is maximum cost of BF process execution time and vertical link reconfiguration.

$$\tau_2 = \begin{cases} A & ,if\ col = 1 \\ B & ,if\ cols = 2 \\ C & ,if\ cols = 5 \\ D & ,if\ cols = 10 \end{cases} \tag{8}$$

$$\tag{9}$$

τ_3 is the time to reload data memory for *vcp* processes running in current column of tiles. The same logic used for τ_6 can reduce τ_3 to zero which we have not considered.

$$\tau_3 = t_d \times \begin{cases} 2 & ,if\ cols = 1 \text{ or } 2 \\ 1 & ,if\ cols = 5 \\ 0 & ,if\ cols = 10 \end{cases} \tag{10}$$

τ_4 is the time to run all vertical *copy* processes during application lifetime.

$$\tau_4 = t^r_{cp} \times \begin{cases} 3 & ,if\ cols = 1 \text{ or } 2 \\ 2 & ,if\ cols = 5 \\ 1 & ,if\ cols = 10 \end{cases} \tag{11}$$

τ_5 is the time to configure horizontal links from each column to the next column.

$$\tau_5 = t_l \times cols \tag{12}$$

τ_6 is the time to reconfigure data memory for *hcp* process.

$$\tau_6 = 0 \tag{13}$$

τ_7 is the time to send results from current column to next column or to the output.

$$\tau_7 = t^r_{cp} \tag{14}$$

3.3 Performance versus Area and reLink cost

To evaluate our 1024-point Radix2 FFT implementation on a matrix of tiles, we implemented FFT in different number of columns subject to the fact that that there are 8 tiles in each column. Figure 10 and 11 illustrate the relation between number of tiles used in a design and link reconfiguration cost with throughput. As depicted in Figure 10,

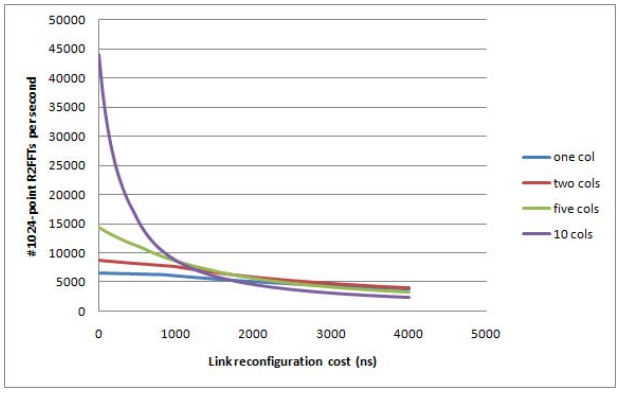

Figure 10: Execution Time for 1024-point Radix2 FFT

using more columns gives a better performance when link reconfiguration cost is small. For example using 10 columns to implement 1024-point Radix2 FFT, in best case gives throughput of around 45000 1024-point Radix2 FFTs per second, while throughput in a high end PC computer is roughly 1000.

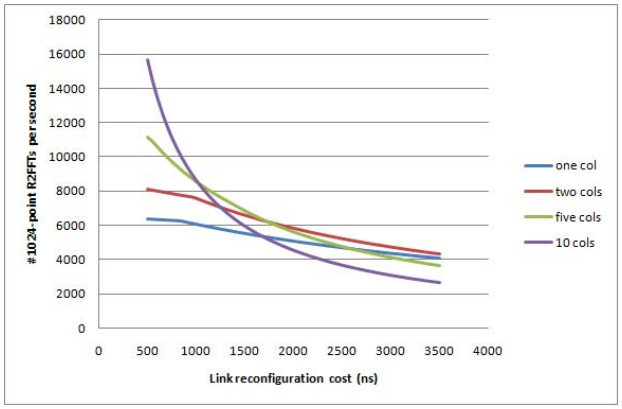

Figure 11: Interesting Part of Figure 10

Figure 11, which is a zoom on the interesting part of Figure 10, illustrates that circuits with more columns are more sensitive to link reconfiguration cost. Throughput is defined as number of 1024-point Radix2 FFTs computed in one second. In Figure 12 the effect of increasing number of columns (number of tiles) on throughput, for different link reconfiguration costs, is depicted. According to the results shown above, while link reconfiguration cost is small, increasing number of columns will increase the performance. However when the link reconfiguration cost exceeds 700ns, increasing the number of columns does not giving noticeable performance. And link reconfiguration cost more than 1100ns has opposite effect on throughput. It may also be noted that effect of link reconfiguration cost is more visible in a design with smaller partition size, also in a design with more number of tiles.

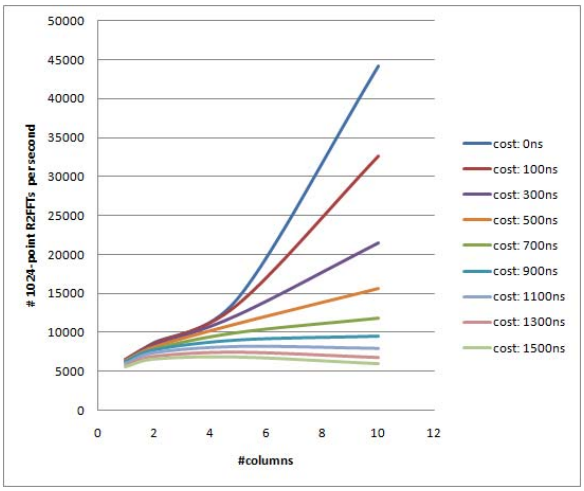

Figure 12: Link cost influence on Radix2 FFT implementation

In the next subsection, we do a design space exploration for the second compute intensive kernel. The JPEG encoder throws up a different challenge because the processes are very compute intensive and we can partition the processes at runtime to obtain the desired performance metrics without encountering too much of the interconnect reconfiguration overhead.

3.4 JPEG Encoder

In this section a brief description of the JPEG Encoder processes is given. Then a manual mapping of these processes to some specific number of tiles is presented. The rebalancing strategies discussed in detail in the next subsection are then applied to this application. The JPEG Encoder consists of five main processes; shift, DCT, quantize, zigzag and huffman. There are some other helper processes which essentially help to perform the mapping efficiently and reduce the execution time. For example, dct is a process which does a blocked computation of the DCT by dividing the original process into four sub blocks.

$$DCT = \left\{ \begin{array}{cc} dct_{00} & dct_{04} \\ dct_{40} & dct_{44} \end{array} \right\}$$

As DCT is the heaviest process of this application, dct is used in some of the implementations to reduce total execution time by about four. CP64, CP32 and CP16 are processes used to copy 64, 32 or 16 elements from a tile to the adjacent tile.

Huffman is the most code intensive process which does not fit in a tile and hence, it is broken into 5 smaller processes: $hman_1, ..., hman_5$. As illustrated in Table 3, there are three columns for data memory usage of each tile; $data_1$ states number of registers used to save fixed data which will be loaded into memory only once, $data_2$ states number of temporary registers and $data_3$ states number of registers which need to be reinitialized each time the related process needs to be executed.

3.4.1 Manual Mapping

JPEG Encoder is implemented on a circuit consisting of 1, 2, 5, 10 or 13 tiles. In Table 4, T_i is used for i^{th} tile. For long running processes with a small number of instructions, it is advantageous to permamently "pin" the instructions to a tile.

Table 3: Cost of reconfiguration (bits)

	$name$	$inst$	$data_1$	$data_2$	$data_3$	$runtime$ $(cycles)$
Main Processes						
p_0	$shift$	11	0	2	9	720
p_1	DCT	62	64	14	13	133324
p_2	$Alpha$	12	64	2	7	720
p_3	$Quantize$	35	64	7	7	1576
p_4	$Zigzag$	65	0	0	0	65
p_5	$Hman_1$	71	0	10	9	7934
p_6	$Hman_2$	56	0	10	6	1587
p_7	$Hman_3$	151	0	43	12	1651
p_8	$Hman_4$	180	0	17	12	2300
p_9	$Hman_5$	109	21	14	17	6823
Auxiliary Processes						
p_{10}	dct	62	64	14	13	33372
$Type_1$ of copy processes: Targeting optimal memory usage						
p_{11}	$CP16$	11	0	2	2	196
p_{12}	$CP16$	11	0	2	2	369
p_{13}	$CP16$	11	0	2	2	720
$Type_2$ of copy processes: Targeting optimal execution time						
p_{11}	$CP16$	17	0	0	0	17
p_{12}	$CP16$	33	0	0	0	33
p_{13}	$CP16$	65	0	0	0	65

Table 4: JPEG Encoder: Manual Mapping

	$Impl_1$	$Impl_2$	$Impl_3$	$Impl_4$	$Impl_5$
# tiles	1	2	10	13	5
p_0	T_0	T_1	$T_0(f)$	$T_0(f)$	T_4
p_1	T_0	T_0	$T_1(f)$	$---$	$---$
p_2	T_0	T_1	$T_2(f)$	$T_5(f)$	T_4
p_3	T_0	T_1	$T_3(f)$	$T_6(f)$	T_4
p_4	T_0	T_1	$T_4(f)$	$T_7(f)$	T_4
p_5	$T_0(f)$	$T_1(f)$	$T_5(f)$	$T_8(f)$	$T_4(f)$
p_6	T_0	T_1	$T_6(f)$	$T_9(f)$	T_4
p_7	$T_0(f)$	$T_1(f)$	$T_7(f)$	$T_{10}(f)$	$T_4(f)$
p_8	T_0	T_1	$T_8(f)$	$T_{11}(f)$	T_4
p_9	$T_0(f)$	$T_1(f)$	$T_9(f)$	$T_{12}(f)$	$T_4(f)$
p_{10}	–	–	–	$T_{1-4}(f)$	T_{0-3}
p_{11}	–	–	–	$T_{2-3}(f)$	T_{2-3}
p_{12}	–	–	–	$T_1(f)$	T_1
p_{13}	–	T_1, T_1	$T_{0-9}(f)$	$T_{0-12}(f)$	T_{0-4}
Time(μs)	419	334	334	84	86
Avg. Util.	1	0.62	0.12	0.37	0.98
images	2.98	3.74	3.74	14.88	14.43
reconfig.	yes	yes	no	no	yes
reLink	no	no	no	yes	yes

Label (f) is used for such a process. The execution times have been obtained with a 400 MHz operating clock and a 180 MB/s reconfiguration speed using the ICAP interface for images of sizes 200×200 pixels.

According to Table 4 for the first implementation only one tile is used. In second implementation out of two tiles, one is considered for DCT. In third implementation 1-1 mapping is applied to map each process to a separate tile. In fourth implementation also one-to-one mapping is used except for DCT which is broken into four dct processes and mapped to four tiles. And finally in the last implementation four tiles are used the same way as previous implementation and all other processes are mapped into only one tile.

The total time taken to compress a sequence of images, average PE utilization and number of images compressed in

one second are given in Table 4. In computing total time, overhead incurred due to copy operations is also accounted for. Besides that, in an implementation, whether reconfiguration cost or link establishment is needed or not is also depicted. According to this table, whether we use two tiles or 10 tiles, throughput is the same, similarly when we use 5 or 13 tiles. The reason is that in these implementations we do not break DCT to smaller sizes. Therefore whether number of tiles is 5 or 13 (same for 2 and 10), the tile allotted to do dct operation will dominate in determining the system throughput. The best processor utilization among these five implementations is when we use five tiles, which is about 98 percent.

3.5 Mapping Processes to Tiles

Mapping the process network to an array of tiles present in a CGRA like reMORPH requires careful balancing of the pipeline to take care of the compute/communication metric. In this section we propose two rebalancing methods which are used to distribute processes of an application among the tiles of the array efficiently. To apply these algorithms, we start with the process network of the application annotated with some parameters for each process, viz., data memory and instruction memory usage and runtime. In the rebalancing algorithms, we follow an incremental approach, starting with one tile and increasing the number of tiles to a maximum subject to area/power constraints while trying to minimize the objective function which could be area or throughput at each step.

In the first method (reBalanceOne), we start with only one tile. Then in each step we allocate a new tile to the the heaviest tile. The heaviest tile is the tile with maximum execution time which is defined to be the sum of runtime and reconfiguration time for all the processes executing in that tile. The steps are explained with reference to Figure 13.

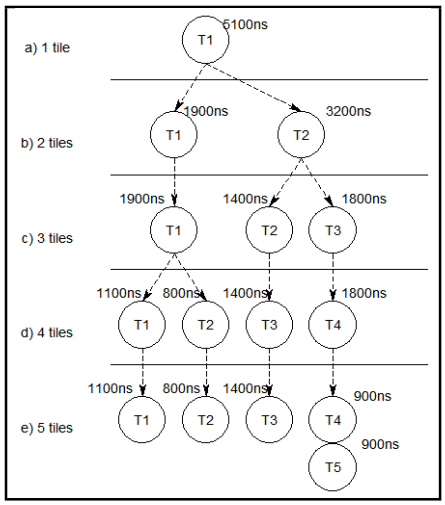

Figure 13: reBalancing

If the heaviest tile consists of more than one process (case a, b, c in Figure 13), the first process is allocated to T_i and the rest of the processes go to T_j. We iteratively move one more process from T_j to T_i if we find a decrease in the total execution time else the previous allocation is retained. If the heaviest tile contains only one process (case d in Figure 13), we create a new tile as an additional instantiation of this heavy tile. The steps are formally specified in Algorithm 1.

ALGORITHM 1. **reBalanceOne**

Require: n : *maximum number of tiles*
  Initially consider one tile: T_1
  Calculate total execution time: $Time(T_1)$
  $\triangle 1 = Time(T_1)$
  while *tiles* $< MaxTilesInDevice$ **do**
    Insert new tile: T_2
    if T_1 *has only one process* **then**
      make T_2 **as a copy of** T_1 ▷ **Total execution time of** T_1 **is now divided into two for** T_1 **and** T_2
    else
      $\triangle 2 =$ **Time**(T_1)
      Move all processes from T_1 **to** T_2
      repeat
        $\triangle 1 = \triangle 2$
        move first process from T_2 **to** T_1
        Calculate $\triangle 2 = |Time(T_2) - Time(T_1)|$
      until $\triangle 2 < \triangle 1$
      move last process from T_1 **to** T_2
    end if
  end while
  return

end

The above algorithm follows a greedy approach which can now be refined to further rebalance allocation of processes to tiles to decrease total execution time of the application. To allocate a new tile, the set "surrounding" the heaviest tile is computed. This set is bounded from left/right by the first tile (in leftside/rightside of the heaviest tile) which has more than one copy (instantiation) or is the first/last tile of the whole circuit.

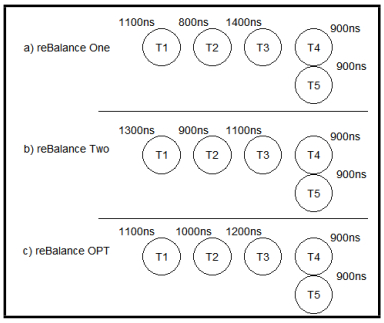

Figure 14: Allocating more tiles to Case e of Figure 13

For example in case (a) in Figure 14, which is the same as case (e) in Figure 13, the set surrounding the heaviest tile (T_1), is $\{T_1, T_2, T_3\}$.
Then the improved algorithm reBalanceTwo will try to redistribute processes among the tiles of this *set* according to the logic mentioned in following paragraphs.

In reBalanceTwo algorithm, we calculate the average execution time for the *set* and then distribute processes of the *set* among tiles such that total time for each tile is close to the average execution time. We continue calculating average execution times and reallocating processes among the set's tiles. Case b in Figure 14 uses algorithm reBalance to redistribute processes among the *set* and we can see that the total execution time of the application is reduced from $1400ns$ into $1200ns$. The algorithm is formally stated below.

ALGORITHM 2. **reBalanceTwo**

Find the heaviest tile: T_h
$(Set, m') = Surrounding(T_h)$ ▷ Set **is the set of tiles surrounding** T_h **and** m' **is number of tiles in** Set
Calculate total execution time $Time = 0$
for all tiles in Set **do**
$Time$ += $Time(T_i)$
end for
$AvgTime = Time/m'$
i=0 ▷ **Index of the first Tile in the Set**
j=0 ▷ **Index of first process in Set**
for T_i **in** Set **do**
if $(Time(T_i) + allocate(T_i, p_j)) \approx AvgTime \pm \Delta$ **then**
allot p_j **to** T_i
Increase i **and** j **by one**
else
break
end if
end for
if $Arrangement_{new} \neq Arrangement_{old}$ **then**
goto line 1
else
return
end if
return

end

There is also a third approach which is the Optimal approach where we find all possible distributions of processes among the tiles of the *set*, and choose the one which gives the minimum total execution time for the *set*.

3.5.1 Automated Mapping

Revisiting the JPEG Encoder, instead of breaking a heavy process into smaller processes and distributing them among some tiles as done in the previous implementation, we can duplicate and instantiate the heavy process into more than one tile. This allows us to propose "*rebalancing*" algorithms which allocate processes to tiles in an efficient/(sub)optimal manner. In the left half of Figure 15, DCT is the heaviest process of JPEG Encoder application — so more than one tile can be used for it and all of them do the same operation, but in a pipelined manner. Therefore, besides memory reconfiguration, "*reLink*" establishment should be considered for links coming/going to/from different DCT tiles at the proper instances of time.

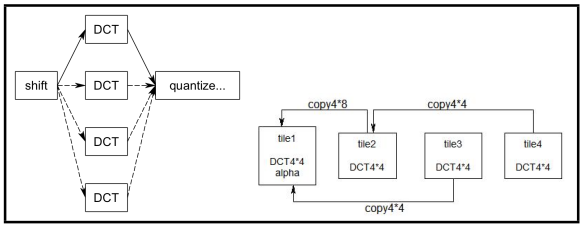

Figure 15: Instantiating a tile n times for a heavy process

Communication pattern between tiles doing dct in third and fourth implementations is also illustrated in the right half of Figure 15.

Table 5, gives implementation details while using first rebalancing algorithm for a circuit of size 24 tiles. In this table, overhead of copy operation from one tile to the other tile (cp64) is not considered because it just adds a fixed amount of time to total time taken to process a block of image. Number in parenthesis states the number of tiles instantiated for the mentioned process. $p_1(17)$ and $p_5(2)$ in column "T_2" and "T_4" mean that 17 tiles are instantiated for process p_1, and two tiles for process p_5. And p_{2-4} in column "T_3" means that only one tile is used for processes p2 to p4.

Table 5: Binding processes to 24 tiles

tiles	T_1	T_2	T_3	T_4	T_5	T_6	T_7
24	p_0	$p_1(17)$	p_{2-4}	$p_5(2)$	p_6	p_{7-8}	p_9

As depicted in Figure 16, applying proposed reBalancing algorithms gives the same mapping in most cases. The reason is that in most cases the heaviest tile contains only one process and after breaking it into two tiles, these tiles still remain the heaviest tiles. As mentioned in "reBalanceTwo" algorithm the processes will be reallotted to the tiles in a set which contains the heaviest tile, but this heavy tile contains only one process and hence no further rebalancing is possible by these algorithms. Therefore three mentioned algorithms give the same mapping and hence the same total execution time and throughput. However, in a few cases, when the number of tiles is between 16 and 20, the heaviest tile contains more than one process, Therefore "reBalanceTwo" and "reBalanceOPT" which is an optimal allocation strategy show interesting results.

Figure 17 illustrates average tile utilization after reBalancing for circuits using different number of tiles in the range 1 to 25.

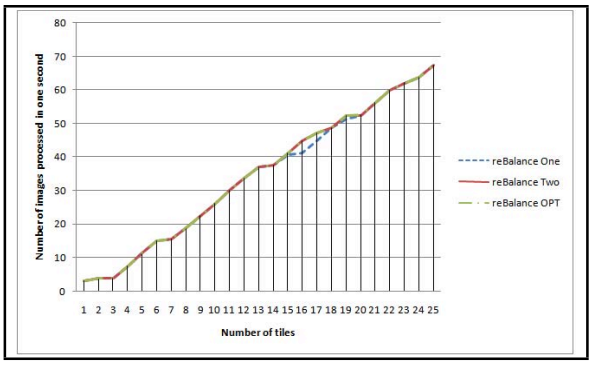

Figure 16: Average PE utilization for JPEG Encoder application

4. RELATED WORK

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)*.

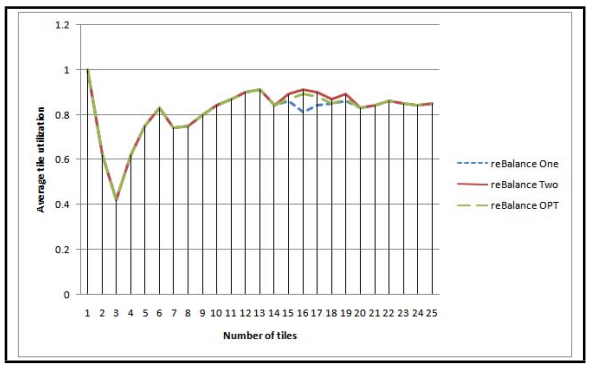

Figure 17: Average PE utilization for JPEG Encoder application

Lysaght et. al. [3] developed an early system for partial reconfiguration using CPLDs for exploiting RTR. An early fine grained dynamically reconfigurable fabric called GARP [4] was developed at Berkeley. To overcome the overhead of serial programming of single context FPGAs many multi context devices and architectures have been proposed in the last fifteen years. Dehon et al [5] introduced DPGA which stored four 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 including Piperench [8], RAW [9], Montium / Chameleon [10], Imagine [11] etc. The FFT class of algorithms has been widely studied in the last half century. High performance implementations of the FFT use variants of the Cooley Tukey method [12]. Duhamel et al. present an excellent survey of Fast Fourier Transforms [13]. A lot of techniques have been used to design efficient hardware implementations of 1D and 2D FFT [14, 15].

5. CONCLUSION

Partial runtime reconfiguration has not been exploited for building high performance applications primarily because of the reconfiguration overhead. Active partial reconfiguration can be effectively utilized using a coarse grain architecture to rebalance the pipeline. A design space exploration methodology to build high performance designs by exploiting partial reconfiguration has been outlined. Two compute intensive kernels have been mapped into a CGRA and performance figures for various cost metrics have been presented. In the future, we will develop a formal process network formulation for performing an automated mapping, placement and dynamic routing for applications in the signal processing domain.

6. REFERENCES

[1] K. Paul, C. Dash, and M. S. Moghaddam, "**reMORPH** – A Runtime Reconfigurable Architecture," in *DSD*, 2012.

[2] M. Liu, W. Kuehn, Z. Lu, and A. Jantsch, "Run-time partial reconfiguration speed investigation and architectural design space exploration," in *In Proc. of the International Conference on Field Programmable Logic and Applications*, 2009.

[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] 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.

[9] 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.

[10] 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 Ubiquuitous Computing, HUC'99*, 1999.

[11] 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.

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

[13] P. Duhamel and M. Vetterli, "Fast fourier transforms: a tutorial review and a state of the art," *Signal Process.*, vol. 19, pp. 259–299, Apr. 1990.

[14] A. Al Sallab, H. Fahmy, and M. Rashwan, "Optimized hardware implementation of fft processor," in *Design and Test Workshop (IDT), 2009 4th International*, pp. 1 –5, nov. 2009.

[15] N. Miyamoto, L. Karnan, K. Maruo, K. Kotani, and T. Ohmi, "A small-area high performance 512-point 2-dimensional fft single-chip processor," in *Solid-State Circuits Conference, 2003. ESSCIRC '03. Proceedings of the 29th European*, pp. 603 –606, sept. 2003.