A DYNAMIC RECONFIGURABLE FABRIC FOR PLATFORM SOCS

C. Papachristou, J. Weaver, R. Vijayakumar and F. Wolff

Department of Electrical Engineering and Computer Science
Case Western Reserve University
Cleveland, Ohio 44106

1. ABSTRACT

A dynamic coarse-grained reconfigurable architecture which targets computationally intensive applications like multimedia and wireless applications is presented. Fundamental features of the architecture are an interconnection matrix, switch buffers, functional units, and a controller. Important aspects of the architecture is dynamic datapath reconfiguration over each control step of the application. Simulations of several classes of applications with results are also presented. Results of Xilinx Virtex 4 Implementations are provided.

2. INTRODUCTION & MOTIVATION

Motivation. VLSI chip functionality has traditionally concentrated into two device categories, microprocessor and ASIC (Application Specific Integrated Circuits). ASICs have high performance but are rigid addressing a specific application, whereas microprocessors are flexible for general-purpose applications. A third category, FPGAs (Field Programmable Logic Arrays) provide, through programmable logic, good flexibility but lower performance. However, present system on chip (SOC) technology pulls together ASICs, microprocessors and FPGAs into a single polymorphic multiple core chip design. SOC technology is well suited for many new compute-intensive applications such as signal and image processing, visualization, wireless communications, and networking.

There is an important need for reconfigurable hardware in future SOC applications. For example, SOC products will be wireless communicating with powerful servers for applications. Optimizing performance while maintaining low power consumption will depend on the SOC ability for quick or even dynamic reconfiguration.

There are two emerging trends for SOC configurability. a) Platform FPGAs which integrate into a large FPGA structure microprocessor cores, ASIC blocks (e.g. hardwired multipliers) and memories. b) Platform SOC integrating microprocessor, ASICs and memory cores but also reconfigurable hardware fabrics. Platform FPGAs are reconfigured using the FPGA structure; however, configuration of Platform SOCs can be done by the reconfigurable hardware. Platform FPGAs have the advantage of platform stability, but they are tied to the vendor's offerings. It appears that Platform SOC fabrics are more suitable for implementing multiple core systems because they are more flexible, potentially can consume less power, and are amenable to dynamic or even autonomous reconfiguration. In such an environment, a reconfigurable datapath core could be driven to data intensive applications whereas a microprocessor core to other functions.

Contribution. In this paper we present a new coarse grain reconfigurable architecture fabric based on a switch buffer matrix connecting functional unit blocks that operate in parallel. The fabric is well suited for integration into a platform SOC of a multicore system, particularly to accelerate computation intensive applications. The fabric is scalable in terms of functional units but also at the bit level in that functional units can be bit-sliced using and connected through the same switch matrix. We also present a design synthesis technique for mapping applications into the reconfigurable fabric. The mapping has been implemented and tested using several High Level Synthesis tools as well as new tools we developed. We have prototyped our architecture into the Xilinx Virtex IV platform, including simulation and implementation of benchmark applications.

3. RECONFIGURABLE ARCHITECTURE

The basic idea of the reconfigurable fabric is a distributed set of programmable processing tiles that are capable of instantaneous dynamic reconfigurability, Fig. 1. A *tile* consists of three types of hardware units or resources, i.e. functional unit, a distributed switch buffer matrix, and local control unit. All hardware resources are connected together through a loop of bus-line interconnects. The functional units are configurable to perform basic arithmetic/logic functions such as Add, Subtract, Multiply. The controller is normally fine grain so it can be implemented using conventional FPGA technology. The routing of data between the switch buffers and functional units are carried out by the switch buffer matrix, e.g. a sort of cross bar, which is embedded in the tile. A programmable tile goes much beyond the current FPGA technology. A tile achieves a middle grain configuration by efficiently allocating its resources, i.e. functional units and switch buffers as well as their interconnects. Configuration occurs within a tile and along several tiles, which can be interconnected into a reconfigurable fabric.

Basically, tiles are suitable for efficiently implementing application function modules such as the FIR filter, FFT, DCT and convolution coder. There are two related problems that have been addressed in this research: (i) mapping a function module into a tile; (ii) reconfiguring dynamically a tile for another function module. For the mapping problem, we propose to use the following 2-phase design process: a) data flow transformation of the function description (e.g. C code) into a resource scheduled graph, b) allocation of data flow elements into the tile hardware resources (operators, cache and interconnects, Fig. 1).

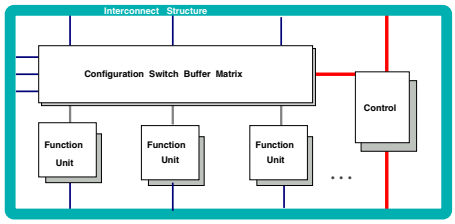

Fig. 1. Reconfigurable Tile

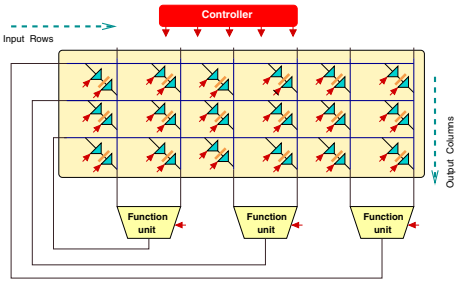

Fig. 2. Dynamic Reconfiguration Switch Buffer Matrix

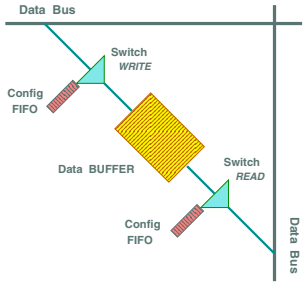

Fig. 3. Parallel Switch Buffers

The above process is in a way similar to the microarchitecture synthesis process, thus we intend to leverage our extensive work we have done in this area, [1]. Using the above process, we can derive the configuration matrix which is a time vs. resource chart describing the mapping. Clearly, the configuration matrix can be precomputed to be readily available for dynamic loading. Loading the configuration matrix into the tile initiates the application.

One important feature that distinguishes our proposed reconfigurable hardware from FPGAs and DSPs concerns the ability to configure the hardware datapath bit-length. We use a bit-slice approach to build flexible bit-length functional units together with their interconnects. Thus applications that demand unusual bit lengths such as 22 bits or 36 bits will be accommodated by dynamically configuring tiles to match these bit lengths. This is a real advantage of our proposed tiles with respect to both FPGAs and DSPs, because one would need a fixed 32-bit DSP to accommodate applications with irregular 22 bit-length, and would be unable to do 36 bits. At the same time, to scale an FPGA to that bit-length may require using far away logic blocks in the chip incurring delay overheads.

The reconfigurable hardware fabric is assembled by hierarchically connecting tiles into a tree structure with the tiles being the leaf nodes of the tree. This hierarchy provides good scalability of the fabric for expansion. It is important both for mapping and for dynamic reconfiguration of tiles within the fabric. Idle tiles can be turned off to reduce power.

Dynamic Configuration

We use a switch matrix in every tile to interconnect the functional units and I/O channels, Fig. 2. Every row is connected to an functional unit output or input channel, and every column to an functional unit input or output channel. I/O channels are not shown in detail. The matrix rows and columns are connected only through switching elements shown in Fig. 2. Actually each switching element consists of 2 switches and a FIFO data buffer queue in between. This accommodates the READ and WRITE phases of each functional unit, respectively. More details are in Fig. 3, where the first switch enables a buffer WRITE and the second a buffer READ, respectively.

The data buffer queues hold variable values coming from a source functional unit, stored for the lifespan of the variable, and used by the destination functional unit. A buffer queuen can hold several variable values in sequential order. The role of the switches is to enable WRITE/READ datapath segments. The switch matrix mechanism in Fig. 2 avoids WRITE conflicts because every WRITE switch in Fig. 2 is associated with a unique functional unit output. However, READ buffer conflicts are possible. This occurs when two variables with conflicting lifespans are stored in the same buffer. These conflicts can be avoided by a) scheduling and allocating the application operations under the above lifespan constraint; b) use a prioritized buffer mechanism based on the READ time tag of the stored variables. The details are discussed in the next section. Buffer overflow needs to be investigated further.

In addition to the data buffer, each WRITE and READ switch has also a control buffer, or FIFO, which holds WRITE and READ configuration information, respectively. Every FIFO bit corresponds to a control step of the particular application running in the tile. Note that if the units are multifunctional, then additional control FIFOs are needed for each functional unit, in like manner to the switch FIFOs of Fig. 3. The proposed scheme allows for variable latency operations, however, latency information should be reflected in the FIFO bitstreams of the switches.

Shown in Fig. 4 is a proposed approach to dynamic configuration. Assume that an application has been precompiled and loaded into the Configuration memory as a binary. Note that the Configuration memory can be structured into long words with each word corresponding to the entire switch matrix of a tile. Further, each long word is partitioned into data fields where each field corresponds to a particular switch FIFO of the matrix. Note, if the application demands longer execution time, then we may need several long words in the Configuration memory for the application. Configuring the application on the tile amounts to shifting the data fields into the FIFOs as shown in Fig. 4.

4. APPLICATION MAPPING TECHNIQUE

We begin with an initial representation of an application in a hardware design language such as VHDL. We parse the VHDL into an intermediate representation, i.e. a Control/ Data flow graph (CDFG) and then schedule the CDFG using a scheduling method and tool that we have developed in our earlier work [17]. Note that for the VHDL parsing and CDFG scheduling there are many well known methods and tools developed in High Level Synthesis works [18] that can be used as well.

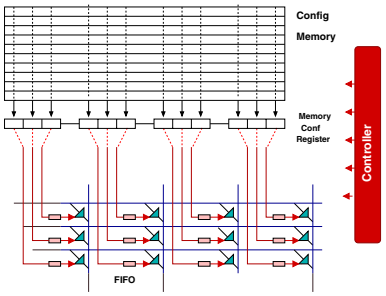

Fig. 4. Configuration Memory

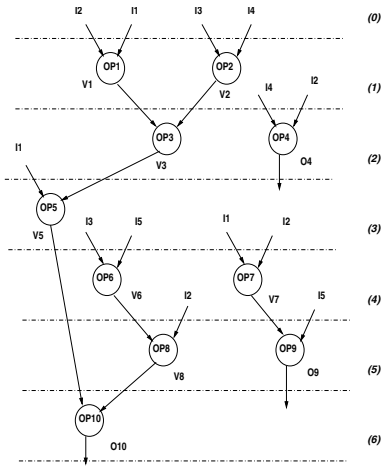

Fig. 5. Simple SCDFG

One of our contributions in this paper is a Resource Allocation technique mapping the operations and variables of the scheduled CDFG (SCDFG) into the architecture resources, i.e. the functional units (FU) and the switch buffers, respectively. We describe our mapping method briefly using a simple example of a SCDFG, Fig. 5.

Our mapping consists of two phases. In the first phase, we allocate functional units and data buffer queues. In the second phase, we use allocation transformations to resolve variable conflicts, if any, and minimize resource costs.

Phase I. Resource Allocation

1) Functional Units. The SCDFG is used to assign each scheduled operation, OPi, to a functional unit, FUj, that can perform the specific task at the assigned timestep, T(OPi). The results of the functional unit allocation for our example are shown in Table 1. The first column indicates the timestep at which an operation was assigned. The second and third columns show the list of operations allocated to each of the functional units, FU1 and FU2, in the matrix. After the functional unit allocation, the inputs and outputs of each operation in the SCDFG are used to allocate the data buffer queues in the switch buffer matrix.

2) Data Buffer Queues. As described above, data buffer queues hold temporary data generated from functional units to be used as inputs to other operations. If an output of a scheduled operation is a temporary variable, it is allocated to the data buffer, DBi,j,k, where i is the functional unit that produces the temporary data, j is the functional unit that consumes the temporary data, and k is the corresponding functional unit input, L(eft) or R(ight). For example, the operation, OP6, in Fig. 5 produces V6, which is a temporary variable. Because OP6 is allocated to FU1 and the temporary variable is used for the left input of

Table 1. Initial FU Alloc

T	$\mathbf{FU}_1$	$\mathbf{FU}_2$
1	OP_1	OP_2
2	OP_3	OP_4
3	OP_5	
4	OP_6	OP_7
5	OP_9	OP_8
6		OP_{10}

Table 2. Alloc after Move

T	$\mathbf{FU}_1$	$\mathbf{FU}_2$
1	OP_1	OP_2
2	OP_3	OP_4
3	OP_5	
4	OP_6	OP_7
5	OP_9	OP_8
6	OP_{10}	

OP8, which is allocated to FU2, V6 is allocated to DB1,2,L.

Table 3. Initial Data Buffer Allocation

T	$\mathbf{DB}_{1,1,L}$	$\mathbf{DB}_{1,1,R}$	$\mathbf{DB}_{2,1,L}$	$\mathbf{DB}_{2,1,R}$	$\mathbf{DB}_{1,2,L}$	$\mathbf{DB}_{1,2,R}$	$\mathbf{DB}_{2,2,L}$	$\mathbf{DB}_{2,2,R}$
1	V_1			V_2				
2	V_1	V_3		V_2				
3		V_3			V_5			
4			V_7		V_5, V_6			
5			V_7		V_5, V_6			V_8
6					V_5			V_8

Table 3 shows the allocation of the data buffer queues for the SCDFG in Fig. 5. The first column indicates the timestep at which the variables are allocated. The remaining columns each represent a data buffer queue in the matrix and show the variables that are allocated to each. Phase I terminates when all operations and their corresponding inputs and outputs are allocated to the switch buffer matrix.

Phase II. Conflict Resolution

After Phase I, the initial allocation is reviewed for read conflicts within the switch buffer matrix. A read conflict occurs when the lifespan of one variable, L(Vi), fully covers the lifespan of one or more other variables, L(Vj)..., in the same buffer queue. The initial allocation in Table 3 shows a read conflict in data buffer, DB1,2,L. The variable, V5, has a lifespan, L(V5), from timestep 3 through timestep 6, whereas, the variable, V6, has a lifespan, L(V6), from timestep 4 through timestep 5. A read conflict occurs because L(V5) fully encompasses L(V6). To solve read conflicts, three transformations are used:

A) Swapping Inputs. This transformation swaps the inputs of a commutative operation by moving the left input to the right input and vice versa. Each input will then belong to a new buffer queue with the prerequisite that no read conflicts occur after the swap.

Table 4. Swap Transformation

T	...	$\mathbf{DB}_{1,2,L}$	$\mathbf{DB}_{1,2,R}$	$\mathbf{DB}_{2,2,L}$	$\mathbf{DB}_{2,2,R}$
1					
2					
3			V_5		
4		V_6	V_5		
5		V_6	V_5	V_8	
6			V_5	V_8	

Table 4 shows the new data buffer queue allocation after this transformation is used to solve the read conflict created by V5. V5 moved from DB1,2,L to DB1,2,R, and V8 moved from DB2,2,R to DB2,2,L. Both V5 and V8 are inputs to operation,

OP10. However, the buffer queue cost is increased by one because V5 moved to an empty data buffer queue.

B) Moving Operations. This transformation moves an operation, OPi, from its current functional unit, FUj , to another functional unit, FUk. If FUk is allocated to another concurrent operation, OPc, then OPc moves to FUj . The prerequisite is that no read conflicts occur after the move. Tables 2 and 5 show

Table 5. Move Transformation

T	$DB_{1,1,L}$	$DB_{1,1,R}$	$DB_{2,1,L}$	$DB_{2,1,R}$	$DB_{1,2,L}$	...
1	V_1			V_2		
2	V_1	V_3		V_2		
3	V_5	V_3				
4	V_5		V_7		V_6	
5	V_5		V_7	V_8	V_6	
6	V_5			V_8		

the new data buffer queue and functional unit allocation after the read conflict fromV5 is solved by this transformation. OP10 was reallocated to FU1 and its inputs, V5 and V8, moved from DB1,2,L to DB1,1,L and from DB2,2,R to DB2,1,R, respectively. The buffer queue cost is decreased by one because both V5 and V8 moved to previously allocated data buffer queues.

C) Moving Predecessors. This transformation moves the predecessor of an operation, P(OPi), to another functional unit. This transformation is similar to (B), but it has the effect of moving data to other buffer queues not reached by combining (A) and (B).

Phase II terminates once the three transformations described above are applied to all read conflicts in the initial resource allocation.

5. RESULTS

The Mapper Program, written in Java, integrates the SYNTEST Parser and Scheduler, developed in our earlier work [1], with the new Resource Allocator and Conflict Resolver algorithms. A standard VHDL description file is given to the Parser, which generates an Unscheduled Data Flow Graph (UDFG). The UDFG is then passed to the Scheduler which produces a Scheduled Data Flow Graph (SDFG). The Resource Allocator and Conflict Resolver algorithms then use the SDFG to allocate resources onto a switch buffer matrix. After the allocation is completed, a bit-level microcode is generated from the control bit strings created from the buffer queue allocation tables. The microcode is stored in a binary file that is used for simulation and synthesis of the FPGA and ASIC implementations.

We implemented the architecture both in Xilinx Virtex IV and derived results on several benchmark applications. We describe the architecture and its basic components in VHDL and use configuration memories to store the microcode control values for the switch buffer FIFOs and the functional unit opcodes.

The VHDL implementations of the different applications in Xilinx and Synopsys Design Analyzer are carried out on Sun Blade 1000. We used the Xilinx ISE 7.1 tools to target the applications to Virtex 4. We followed two target approaches. In the first approach, we used a fixed bit-width for the application datapath, fixed to 8 bits. In the second approach, we used bit-slicing based on 4-bit architecture slices that can be configured together to form 8-bit, 12-bit and so on structures. We used a clock period of 10 ns for implementation for all the results.

Table 6 shows the results of the Xilinx implementations of the applications using the fixed bitwidth approach. It provides the targeted Virtex 4 part, the critical stage delay, the actual time to perform a schedule step in the application and finally the amount of utilization of LUTs and Slices for each application. The applications are targeted to the least cost Virtex 4 part, in which they would fit. Recall in Fig. 2 the schedule step consists of reading data from data buffers and passing it to the functional units for computations and write step, where the computed delay values are passed to the data buffers for storage. The delay value from Xilinx is the sum of these two values. From this table, we could see that as the number of time steps and the functional units in the application increased, the next bigger part had to be chosen, thereby increasing the cost of the implementation.

Table 7 shows the results of the Xilinx implementations using the bit slice approach. The table gives us the same kind of results as above. Using the bit slice approach, we were able to find that the same applications needed a bigger part to fit the same application, keeping the hierarchy of the architecture. Xilinx tools were able to partition the two 4 bit structures and combine them to produce the 8 bit data width architecture. Since the same application had to be targeted to a bigger part, the cost of implementing the same application went up as compared to the fixed bit data width approach. To test multiple and concurrent applications, we combined three examples, the Bpfilter, DCT and Wave filter, to make a multiple schedule. This schedule was optimized for the number of time steps and functional units then implemented in Xilinx, and the results of the various values are shown in Table 6 under Multiple Schedule.

Table 6. Results in Xilinx using fixed bitwidth approach

Applic.	Virtex 4	Delay	Schedule Time Steps		% Utilization	
			Actual	Xilinx	LUTs	% Slice
Elliptic	xc4vfx20	5.65 ns	10 ns	13.52 ns	63 %	53 %
ArFilter	xc4vfx20	5.71 ns	10 ns	13.15 ns	45 %	44 %
Bandpass	xc4vfx20	6.18 ns	10 ns	11.13 ns	53 %	44 %
BpFilter	xc4vfx20	5.63 ns	10 ns	12.98 ns	58 %	50 %
DCT	xc4vfx20	5.94 ns	10 ns	11.20 ns	55 %	46 %

Table 7. Results in Xilinx using bit slice approach

Applic.	Virtex 4	Delay	Schedule Time Step		% Utilization	
			Actual	Xilinx	LUTs	% Slice
Elliptic	xc4vsx55	6.17 ns	10 ns	15.19 ns	45 %	28 %
ArFilter	xc4vsx55	5.76 ns	10 ns	13.53 ns	36 %	24 %
Bandpass	xc4vsx55	5.96 ns	10 ns	14.52 ns	39 %	24 %
BpFilter	xc4vsx55	6.32 ns	10 ns	13.93 ns	42 %	26 %
DCT	xc4vsx55	6.04 ns	10 ns	13.52 ns	39 %	24 %

6. REFERENCES

[1] C. Papachristou. M. Nourani, "Stability-based algorithms for scheduling and allocation in high level synthesis of digital systems," *IEEE Transactions on VLSI*, Jan. 2001.

[2] Y. Lin, "Recent developments in high level synthesis," *ACM Trans. on Design Automation*, Jan. 1997.