Automatic Accelerator Preemption

Alireza Khadem
Computer Science and Engineering
University of Michigan
Ann Arbor, USA
arkhadem@umich.edu

Sai Rohit Kandula
Computer Science and Engineering
University of Michigan
Ann Arbor, USA
rohitkan@umich.edu

***Abstract*—FPGA accelerators are being used for various applications and have found a place in the cloud and mobile devices. By mapping various contexts/tasks to a single FPGA chip to improve the FPGA utilization and sharing. However, FPGAs inherently do not support context switching and it has to be built into the initial RTL files manually, which increases the development time. Our project is to create a framework to automatically add preemption capabilities to any HDL design.**

We achieve this goal in four stages: *Prepass*, in which we preprocess the design, *Remove FF*, which removes all of the registers from the accelerator, *Add State Module* that adds a new module named *State Module* and keeps all of the design states, and *Make Top Module* in which we introduce a new top module consisting of the previous design and the *State Module*.

To evaluate this idea, we tested on two popular accelerators: First, *Neural Network Accelerator* processes a fully-connected neural network and reads/stores from/to the off-chip memory using Avalon Memory Mapped interface. Second, *Edit Distance Accelerator* which is commonly used in Genome Sequencing. We stick to the dynamic programming approach of this algorithm and implement the accelerator in a systolic-array mode.

We break the *initial* body of our testbenches into different contexts and interleave them on the FPGA. Finally, we synthesize our preemptable accelerators using the Vivado Design Suite and found the overhead of *State Module* to be 34% and 52% on the *Neural Network Accelerator* and *Edit Distance Accelerator*, respectively.

***Index Terms*—Field Programmable Gate Array (FPGA), Hardware Preemption**

I. Introduction

As Moore's law comes to end, CPUs are not able to provide enough throughput for highly-parallel workloads. In this regard, Field Programmable Gate Arrays (FPGAs) and General Purpose Graphics Processing Units (GP-GPUs) have attracted the engineers in the semiconductor industry. FPGA outperforms GPU in terms of FLOPS per watt since it consumes less power and energy. With this in mind, Cloud services like Amazon Web Services have started offering FPGA instances [1] and Apple have added FPGAs to iPhone chipset since iPhone 7 [3].

Unlike Application Specific Integrated Circuits (ASICs), time to market for an FPGA is faster as it involves only writing HDL. However, ASICs have better performance as they are highly optimized. But fixed hardware in ASICs means new update needs a respin of the hardware. FPGAs enjoy a sweet spot of offering better performance than CPUs, being cheaper than ASICs and easy to test, deploy and update. Hence, FPGAs are now found in many smart offering FPGAs capable machines. Recent developments in FPGAs have increased their capabilities.

FPGA is a reprogrammable hardware that is made up of programmable LUTs (LookUp Tables) with a programmable interconnect that can implement any Boolean function. Beside LUTs, DSP units that perform signal processing and MAC operations and on-chip memories like BRAM are the other parts of the today's FPGAs. Recent FPGAs include ARM CPUs too. With the slowing down of Moore's law and advent of new applications like machine learning, genomics etc., FPGAs have found reinvigorating use as accelerators.

Typical toolchain flow involves writing Verilog, testing it on a simulator. Then this is compiled and run through a place and router which maps to the hardware units on the FPGA and optimizes for performance. The final output is a bitstream which is sent to the FPGA by a driver and programs the fuses on the FPGA. Since many tasks can benefit from FPGA acceleration, support for context switching would be very beneficial and can be used for virtualizing across multiple users and tasks. This would improve utilization and lead to better bottom-line margins. In case of smartphones, an FPGA can reprogrammed on the fly and used for a task of higher priority. For example, if a user wants to take a photo, an FPGA accelerator can run some machine learning algorithms to support Bokeh effect and if he switches to voice assistant the same FPGA can be reprogrammed to run ML for speech recoginition. This saves adding different accelerators for different purposes [10]. refer to this as accelerator level parallelism. (XLP). The other kind of simpler use of context switching it simply stopping execution of current input and process another input which could have a higher priority.

In this project, we try to create a framework to automatically add support for preemption to any input Verilog files. We use an open-source RTL synthesis tool called Yosys [5]. Yosys is capable of exploiting user-defined functions called *pass*. we create a pass called *Add Preemption* to detect and duplicate state elements to the number of contexts needed. We tested this pass on two real world accelerator designs, one for machine learning and another edit distance accelerator which is used in genomics. For these testcases, we used Intel Modelsim Simulator and Xilinix Vivado tools for verification and synthesis respectively. We used Xilinx Atrix-7 model in Vivado for synthesis.

Contributions of our project are

- An Yosys pass to detect flip flops in a module and replicate them.
- BRAM library cells with preemption support.
- Proof of concept on two real world accelerators: *Neural Network Accelerator* and *Edit Distance Accelerator*

The following sections of this report are arranged as follows:

In Section II we go through the high level approach we use for accelerator preemption and talk about preemptable BRAM cells. Section III explains four sub-tasks we do to make an accelerator preemptable: *Prepass*, *Remove FF*, *Add State Module*, and *MakeTop Module*. We go through two real-world examples in Section IV that are *Neural Network Accelerator* and *Edit Distance Algorithm*. In section V, we investigate three prior papers that are similar to our idea in spirit. Section VI proposes four ideas to further improve our implementation as the future works. Finally, we conclude in Section VII.

II. Design

A. High level Approach

Operating Systems support context switching by saving the context state like registers, Program Counter and Stack Pointer values etc and loading the state of the next process to be run. Similarly to achieve context switching in a FPGA, we save the output of the flip flops and load inputs of the new context. This state can either be streamed out to system memory or the flip flops can be replicated one for each context. The first approach is efficient in terms of area but context switching is slow whereas the second approach the area overhead is proportional to the number of contexts supported but the context switching can be done in one cycle. We decided to go with approach 2. In this approach, we add a new bus *Selector* to select the context. This is driven by the user.

B. Preemption support to BRAM cells

We also implemented BRAM standard cells with preemption support. The BRAM memory is divided based on the number of contexts and the address space selected using the bus *Selector* which is also used for selecting the context. The input address MSB bits are masked based on the Selector bus. We used 7-series Xilinx FPGA design for our implementation. The Xilinix Atrix 7-series FPGAs has the following BRAM types:

- Dual Port: It has 2 independent access ports for read and write. There are 36Kb (20 in number)and 18Kb (40 in number) variants.
- Simple Dual Port: The dual port BRAMs can be configured as a single port doubling data bus widths to 72Kb and 36Kb.
- Cascading: 2 adjacent 32Kb BRAMs can be cascaded to form a 64Kb RAM.

We implemented single port and dual port and leave cascading BRAM for future work. Instead of modifying the library Verilog files, we created a simple SRAM wrapper cell with parameters address width, data width, size and number of ports. This was then synthesized to use in built BRAM cells using the option `(* ram_style = block*)`. We also tested these blocks independently with small testcases which write into the correct memory space and read them out. Figure 1 shows the BRAM mappings in our implementation.

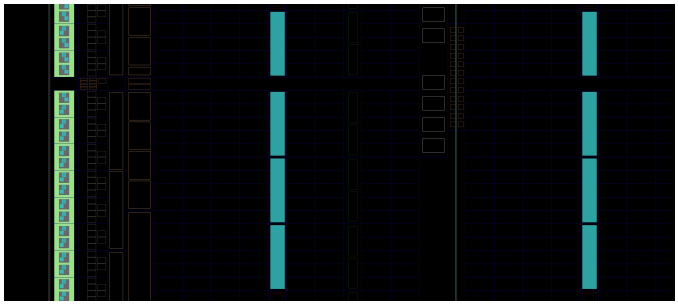


Fig. 1. Layout showing the mapped BRAMs.

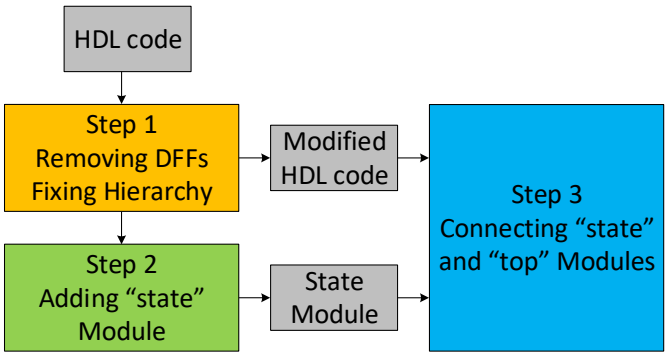


Fig. 2. Highlevel implementation of Accelerator Preemption.

III. Implementation

Users call *Add Preemption* pass in the Yosys command line or bash script with the following command:

```
add_preemption num_of_states clock_name
```

num_of_states is number of contexts we expect the accelerator to accept and clock_name is the name of the clock pin used in the top module file.

We break adding preemption to the design into four individual subtasks which are shown by Figure 2. In the first sub-task, Prepass, we preprocess the design, converting a behavioral code to the structural. Next in the Remove FF, we remove all the flip flops from the design. Make State task aims to make the state module, which adds a module consist of all the states in the design. Finally, we make a new top module and handle the connections between the previous top-module and the state module. Next, we delve into the individual subtasks in the following subsections. The source code of *Add Preemption* pass is accessible in Github [8].

A. Yosys Prepass

Yosys is a RTL compiler which supports both Verilog and VHDL languages. For reading system verilog files, we used

sv2v tool. [4] Like LLVM compiler, Yosys can be extended with user passes. We implement the above mentioned passes in Yosys. Yosys converts an input Verilog file into internal representation called RTLIL and provides APIs to access various components of the design like Modules, Cells and Wires. Before running the passes, we run the following prepasses to infer the flip flop cells in the design.

- *hier*. This pass looks at all modules in the current design and re-runs the language frontends for the parametric modules as needed. We also use this pass to detect the top module of the input design. This piece of information will be used in the *Fix Hierarchy* pass.
- *fsm*. This pass performs FSM extraction and optimization. It detects finite state machines by identifying the state signal and merges additional auxiliary gates into the finite state machine,
- *opt* This pass performs a series of trivial optimizations and cleanups.
  - Identifies wires and cells that are unused and removes them.
  - Performs constant folding on internal cell types with constant inputs.
  - Removes unused inputs from LUT cells (that is, inputs that can not influence the output signal given this LUT's value)
  - Performs various optimizations on memories in the design.
  - Identifies and merges cells with identical type and input signals.
  - Analyzes the control signals for the multiplexer trees in the design and identifies inputs that can never be active. It then removes this dead branches from the multiplexer trees.
  - Identifies flip-flops with constant inputs and replaces them with a constant driver.

We run the *opt* pass after each of the next passes to remove the unused cells and wires in the design.

In the preprocess, we also check the following conditions:

- Input command must be in a correct form, identifying number of required contexts and the name of the clock pin.
- Top module must have been introduced using the hierarchy pass before executing the *Add Preemption* pass. If the top module is not detected in this check, user faces with an error, introducing the *hierarchy* command.
- There must not be any module in the input design under the names of *top_module* and *state_module*. If any pass with these names is in the design, we show and error to the user to change the name of those modules in their design.

B. Pass 1: Remove FF

After running, the basic passes of Yosys, we run remove_ff step which does the following:

- Detect a Flip Flop in a module
- Remove Flip Flop
- Add Cascade ports to the module

For each FF, a pair of input and output ports are added. These ports are later passed to the next pass *add state module*. This pass handles multibit FFs and adds ports of appropriate bitwidth. Another corner case is concatenated signals. Yosys stores them in *sigspec* as chuck and wires are appropriately connected or constants are applied to the correct bits. Floating wires are removed.

After running remove_ff pass, the hierarchy is fixed up. The algorithm is shown in Algorithm 1.

Algorithm 1 Fix Hierarchy

```
boolean noChange ← false
while !noChange do
  noChange ← true
  for List of modules do
    for List of cells in the module do
      if If cell has ports missing in the instantiation then
        Add new cascade ports to hierarchical module
        Connect to the cell's ports
        noChange ← false
      end if
    end for
  end for
end while
```

C. Pass 2: Add State Module

State module holds all of the states removed from the design in section III-B. The information of these states (flip flops) in the system is passed to this task from the Remove FF task. It consists of the name of the input and output wires as well as the width of each state. Besides, user passes the expected number of contexts, *N* to the *add preemption* pass. For each flip flop in the design, we make *N* different states. The input to these flip flops is the input to previous flip flops in the system which is passed to the *State Module*. The output of each state, which is routed back to their modules, is the output of a multiplexer, choosing between different contexts of that state. The selector of the multiplexers is the context selector, which is an input from the *Top Module*. Figure 3 C) shows a simple example of *State Module*.

Yosys implements the reset circuit of each flip flop using a multiplexer, which chooses between 0 and the input to the flip flop. One of the optimization we did on the *State Module* was to introduce a reset pin that resets all of the contexts of states. Prior to adding this pin, reset of *State Module* was handled by the top module reset pin and had to reset flip flops of each context individually. Thus, to reset a system with *N* contexts, we needed *N* clock cycles. With this optimization, we could reset all the context in one clock cycle.

A minor optimization we did on the *State Module* was to accept general clock port name. In the first version of the *Add Preemption* pass, we considered the clock name to be *clock*. As many implementations had different names, we added an

option to the *Add Preemption* pass to take the clock name in the design from the user.

We had two options for the *State Module* implementation:

First, we could write a general module in Verilog or SystemVerilog which holds a configurable number of contexts for a configurable number of modules. Then, in the *Yosys* pass, we could have customize this module and use it as the *State Module*.

The other option we had for this pass was to write the *State Module* from the scratch in Yosys. There are a set of basic cells in Yosys with which developers can write their own modules and add them to the user's code. However, the problem with these cells is that they just provide very basic cells, such as *MUX2to1*. Higher order modules such as a *MUX3to1*, must be written employing the basic cells.

Although the second option seems more challenging, it gives more flexibility for further optimizations. We opted the second choice, and implemented the following modules using the basic cells in Yosys:

- A general *MUXNto1* which chooses between *N* contexts of a state, each has *W* bits.
- A selector decoder which converts the selector into the one-hot representation with *N* bits. The decoded selector is fed into the multiplexers.
- Enable circuits. D-flip flop cell in the Yosys tool only has Clk, D, and Q ports. We had to implement the Enable circuits for each flip flop.
- Global Reset circuit. As mentioned Section III-C, this pin resets all the contexts in the system.

D. Pass 3: Make Top Module

The previous top module and the *State Module* designed in Sec III-C are instantiated in a new module called *Top Module*. We connect the cascade ports of the *State Module* to the previous top module, using the same information of Sec III-B. Other ports of these instances are connected to the new *Top Module*. Figure 3 D) shows this module as well as the cascade connections, inputs, and outputs of the design.

IV. Evaluation

To evaluate our system, we implemented two real-world accelerator examples, a neural network accelerator and an edit distance accelerator. After implementating and testing each accelerator, we ran the *Add Preemption* pass on these designs. We changed the testbenchs to support the preemption. In the new testbenchs, we added and handled the following signals:

- *Global Reset*. Instead of resetting each individual context at the beginning of the simulation, we make this port high for a cycle.
- *Selector*. This pin shows on which context we want the accelerator to work on. We consider two contexts for each of the accelerators. We break and interleave two instances of the *initial* body of the previous test bench. For a concise simulation, we break the new sections from the arbitrary parts and shuffle them in a new *initial* body.

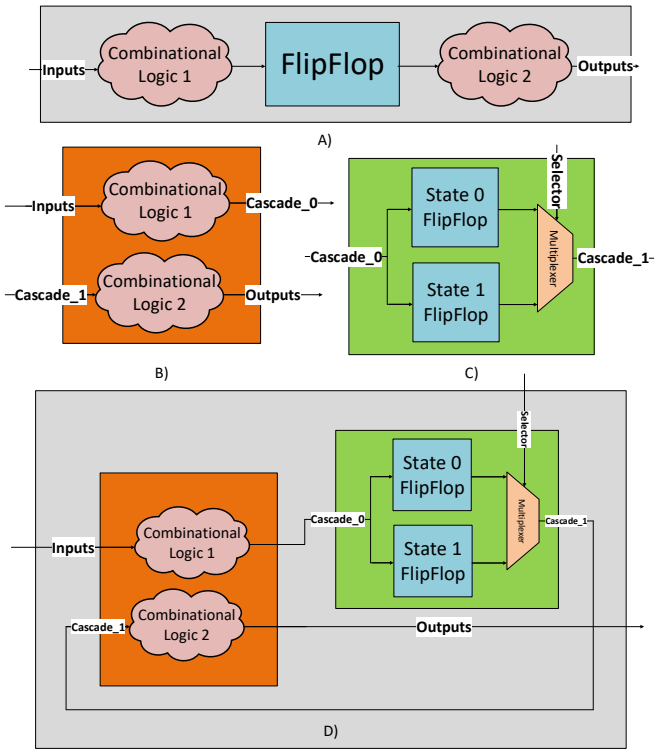


Fig. 3. Three sub-tasks in our design. A) A simple example of a module with 1 flip flop. B) Remove FF eliminates all of the states of the design and adds cascade ports. C) Adding State Module to the Design supporting two contexts. D) Assembling State Module and previous top module into a new module.

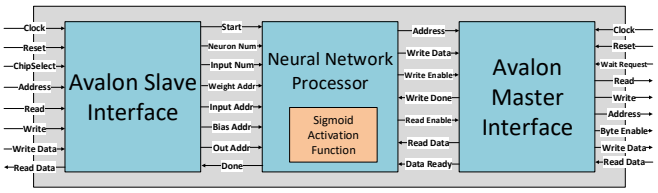
After changing the testbenches, we simulated the designs both using *Intel Modelsim Simulator* as well as *Xilinx Vivado Design Suite*. The final result of each contexts must be consistent with the final result of the individual contexts running on the original version of the accelerators. For calculating area overhead, we used *Xilinx Vivado Design Suite* choosing a Xilinx Atrix 7-series device (XC7A12T).

In the following sections, we go through the two accelerators and the results:

A. Neural Network Accelerator

This accelerator calculates the outputs of a fully-connected neural network with the *sigmoid* activation function. The accelerator is equipped with *Avalon Memory-Mapped Interfaces* which works for Intel FPGAs. The following modules are implemented:

1) *Avalon Memory-Mapped Slave Interface*: The processor connects to the accelerator and sends command through this module. This module has 5 registers, in which the CPU writes and the neural accelerator core reads. The first register shows the number of inputs, number of neurons, if the layer need activation function, a bit that shows the start command, and a bit that states whether the computation is finished or not. The other registers holds the starting addresses of the Weight, Input, Bias,



Slave Address	31	30 – 16	15 – 2	1	0
0	Done	Num of Inputs	Num of Neurons	AF Need	Start
1	Weight Address				
2	Input Address				
3	Bias Address				
4	Output Address				

Fig. 4. Neural Network Accelerator Design. Modules and registers of Avalon Slave Interface.

and Output on the off-chip memory. Figure 4 depicts the contents of the five registers in this module.

2) *Avalon Memory-Mapped Master Interface*: This module is connected to the off-chip memory and reads the weights, inputs, and biases from there. It is also responsible for writing the outputs to the off-chip memory.
3) *Sigmoid Activation Function*: This module calculates the sigmod function. It is implemented using *Maclaurin Series* [2] in 100 iterations.
4) *Neural Network Calculator Core* This module wait for the start command and the layer information coming from the *Slave Interface*. In each iteration, it computes the result of a neuron. It reads and writes the required data through the *Master Interface*. After calculating the Multiply and Accumulate (MAC) operation, this core sends the result to the *Sigmoid* module.

Figure 4 shows the top module of this design. The three modules described above as well as the connections between them are illustrated in this figure. After running the *Add Preemption* pass on this design, 30 states were found and introduced to the *State Module*.

Our testbench executes 2 fully-connected neural network layers with *3 inputs, 4 neurons* and *5 inputs, 3 neurons* configurations. The execution of these layers are interleaved randomly. The final result of these configurations were equal to the final result of the individual execution on the original design.

We used Xilinx Vivado to synthesize the design and compare the overhead. we found the 34% area overhead of our approach on this accelerator for two contexts.

B. Edit Distance Accelerator

Edit distance is a measure of how dissimilar two strings are. It calculates the number of insertions, deletions and gaps between two strings. This is used in many sequence alignment algorithms like Burrows-Wheeler aligner which is used for sequencing DNA. Edit distance uses a 2D Dynamic programming where it recursively computes distance D(i,j) for ith and jth positions of a string. The recursion formula is given below.

$$D(i,j) = min\begin{cases} D(i-1,j)+1 \\ D(I,j-1)+1 \\ D(i-1,j-1)+\begin{cases} 1; if X(i) \neq Y(j) \\ 0; if X(i) Y(j) \end{cases} \end{cases}$$

Systolic arrays are 2D arrays of Processing Elements which are connected together and process data simultaneously and hence exploiting data parallelism. This is a common approach to implement FPGA accelerators.

Edit Distance algorithm can be implemented as an 1D systolic array as shown in figure 5. The accelerator can process 326.65 ops/s whereas CPU does only 50.68 ops/s. This gives a speedup of $\tilde{6}$x over the recursive algorithm running on a CPU. This accelerator had System Verilog files hence we used sv2v tool to convert to Verilog. sv2v tool uses Haskell to map System Verilog constructs to Verilog and supports almost all major features of System Verilog. We ran simulation to check if the correctness is met and tested preemption by stopping one testcase and applying another. We also ran this through implementation step in Vivado and found the area overhead to be 52%. This is expected as flip flops are a significant portion of the accelerator (60% of the combinational) totalling 4000. we are doubling the flip flops to 8000 and adding muxes in the *State Module*. And this design also needs wider muxes and with preemption their usage increases too.

V. Related Works

The idea of adding ports for the flip flops is borrowed from Cascade and we named the new ports as *cascade* in our pass. Cascade is a JIT compiler for FPGAs for speeding up compile-test-debug cycle by mapping Verilog modules into SW engines first and starts compilation at the same time. Once the compilation is done SW engine is moved to FPGA. The communication between SW engine and HW engine is through Cascade runtime and driver. [7]

This paper [6] talks about Partially Re-configurable (PR) FPGAs. FPGA is divided into static and one or more PR regions. The PR regions can reconfigured by downloading a partial bitstream using a device configuration port like Internal Configuration Access Port (ICAP) which reduces the overhead. However this approach needs support in RTL and needs the design to be partitioned into static and PR regions. Our approach needs no modification to the input RTL files.

This paper [9] compares two different approaches in accessing and saving the hardware states. One is through configuration port access (CAP) like [6] and other is having special modules called Task Specific Access Structures (TSAS). In CPA method, the state is saved and restored from configuration memory (CM). The configuration memory is used for storing bitstreams. This approach needs states to be separated and CAP to read and load them. This has poor data efficiency as state is a small part of the configuration frames. TSAS approach uses special structures which are synthesized along with the design to store raw state information to on chip

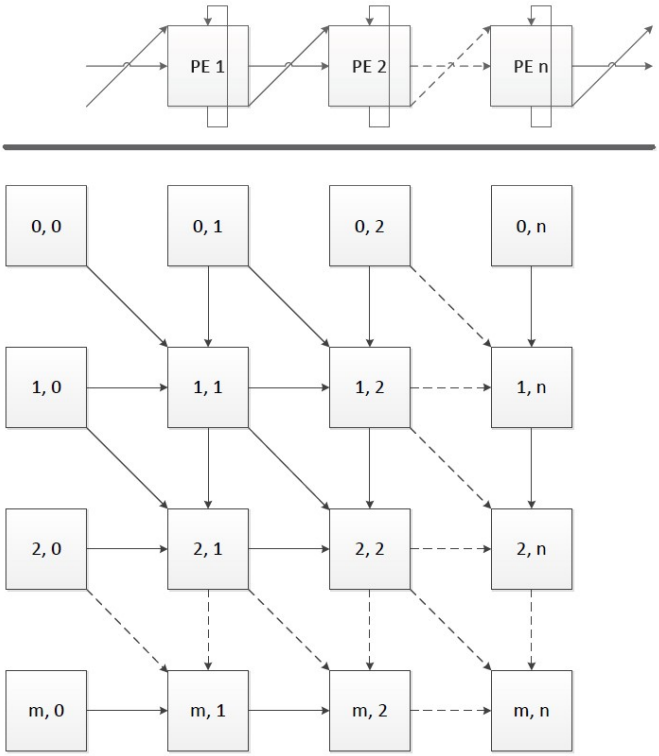


Fig. 5. A systolic array of processing elements (PE) for calculating edit distance. The arrows show the dependencies. The 1D array is the folded systolic array used in the implementation.

or external memory and load it back. TSAS is more data efficient but has high area overhead. Both these approaches need modifications to input RTL. Our approach is somewhat similar to TSAS as we are adding *State Module* which has copies of all the flip flops. The performance of preemption is also closer to TSAS approach.

VI. Discussion

One of the important drawbacks of this idea is the overhead of the *State Module*. This is owing to the fact that memory cells (flip flops) occupy a large area compared to the logic cells. Since the area is critical for large accelerators, we must work on the area overhead of this idea to be acceptable by the community. We propose the following works to be done on this idea:

A. Selective State Recognition

The current implementation of the *Add Preemption* task stores all of the flip flops in the design. As a result, user can switch the context in every possible position of the accelerator run-time. To corroborate this claim we break the *initial* bodies of the testbenches from the arbitrary points and interleave them together in a new testbench. The results proved this claim.

Although we save all of the registers, a great portion of them are used for the temporary values. In the testcase of the Neural Network Accelerator, 46% of the total stored states are used for the temporary registers. In the light of this observation, we propose to store only a portion of the registers that are essential, and force the user to do the context switch whenever the accelerator does not need to the temporary flip flops.

This idea could be implemented by preprocessing the controller of the designs by Yosys. An analysis on the design shows the appropriate cycle that preemption has the least cost. Then we introduce two new signals to the state module to make selective preemption possible:

- *Preemption Request*: This signal is initiated by user, meaning that CPU needs a preemption to be occurred. The specific context to which CPU need is provided by the *select* signal in the current implementation.
- *Preemption Grant*: After getting a request for preemption, *state module* waits for the specific state in the controller FSM that has the minimum cost of preemption. This state is introduced to the *state module* in the preprocessing. When controller switched to this state, preemption happens by changing the context number. *Preemption Grant* is asserted in this situation as an effect of the preemption.

B. Off-Chip Memory State Storage

On-chip memories (LUT, BRAM and SRAM in FPGAs) are counted as a precious resource in the hardware designs. This is owing to the higher speed, larger area, and more energy consumption. Thus, they must be exploited in the best way.

In the current implementation, we store all of the states in the *State Module* in the LUTs of the FPGAs, which are the fastest and the most expensive memories in the FPGAs. As a result, we expect that the area and power consumption of our design to be increased linearly by the number of contexts.

Instead, we suggest to exploit the off-chip memory (DRAM in FPGAs) for lower cost. One context could be stored in LUTs (the current context) while the others are transferred to the off-chip memory. An interface to the DRAM memory must be written (similar to the *Avalon Memory-Mapped Interface* we wrote for the *Neural Network Accelerator*. With this interface, *State Module* is connected to the off-chip memory to store restore the states.

C. State Cache

When the other contexts of the design are stored to the off-chip memory, the latency of context switching increases due to the high off-chip memory access delay. To address this problem, we can exploit the temporal and spatial locality by using a cache to store the recently-used context. *Store Cache* can be designed to use the other on-chip memories, such as BRAM or SRAM.

D. Software Support (Driver)

Although our work covers the hardware implementation of this idea, Software Support is left for future work. A driver would set the selector bus and control contexts and have a scheduler implemented within it. And further the context control can be exposed to the user as an API.

VII. Conclusion

Automatic Accelerator Preemption is proposed in this project. We used a HDL Compiler called Yosys to implement a pass which adds the preemption capabilities to the accelerators. The implements a pass to automatically add the preemption to any input RTL files in four stages: *Prepass*, *Remove FF*, *Add State Module*, and *Make Top Module*.

We developed a preemptable BRAM cell library for 7-series Xilinx FPGAs. This library includes the implementation of a single-port and a dual-port BRAM cell.

The other contribution of this project was running proof of concept on two accelerators. The first one *Neural Network Accelerator* which calculates the result of a fully-connected neural network layer. The other one is the *Edit Distance Accelerator* that processes two strings using Levenshtein distance algorithm. We changed their testbenches to include the preemption simulation. The individual result of two contexts running on these accelerators are compared by the result of their execution on the original designs.

Finally, we discussed the open research areas in this project. The overhead of the preemption capability were around 34% and 52% in the *Neural Network Accelerator* and *Edit Distance Accelerator*, respectively. Three techniques are proposed to reduce this overhead that is a result of *State Module*: *Selective State Recognition* which suggests an architecture for removing the contexts of the temporary registers, *Off-Chip Memory State Storage* and *State Cache* which propose to store the contexts on the off-chip memory and use an on-chip cache for the recently-used contexts.

VIII. Acknowledgements

We would like to thank Jiacheng Ma for his support and guidance throughout the project and helping us out in getting started with Yosys. We would also like to thank Professor Baris Kasikci for his insights and feedback. Finally we also want to thank the others EECS 582 students for their feedback.

References

[1] Amazon ec2 f1 instances. https://aws.amazon.com/ec2/instance-types/f1/.

[2] Sigmoid Function. https://mathworld.wolfram.com/SigmoidFunction.html.

[3] This mysterious chip in the iphone 7 could be key to apple's ai push. https://www.forbes.com/sites/aarontilley/2016/10/17/iphone-7-fpga-chip-artificial-intelligence/.

[4] Tool for converting system verilog to verilog. https://github.com/zachjs/sv2v.

[5] Yosys Open SYnthesis Suite. https://github.com/YosysHQ/yosys.

[6] Rohit Kumar Aurelio Morales-Villanueva and Ann Gordon-Ross. Configuration prefetching and reuse for preemptive hardware multitasking on partially reconfigurable fpgas. *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1505–1508, 2016.

[7] Michael Wei Eric Schkufza and Christopher J. Rossbach. Just-in-time compilation for verilog: A new technique for improving the fpga programming experience. *ASPLOS '19: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 271–286, 2019.

[8] Sai Rohit Kandula and Alireza Khadem. Eecs 582 project. https://github.com/rks90/EECS_582, 2020.

[9] Masato Edahiro Shinya Honda Krzysztof Jozwik, Hiroyuki Tomiyama and Hiroaki Takada. Comparison of preemption schemes for partially reconfigurable fpgas. *IEEE Embedded Systems Letters*, 4(2):45–48, 2012.

[10] Vijay Janapa Reddi Mark D. Hill. Accelerator-level parallelism. *arXiv*, 2019.