A Time-Multiplexed FPGA Overlay with Linear Interconnect

Xiangwei Li*, Abhishek Kumar Jain†, Douglas L. Maskell* and Suhaib A. Fahmy‡
*School of Computer Science and Engineering, Nanyang Technological University, Singapore
†Lawrence Livermore National Laboratory, United States
‡School of Engineering, University of Warwick, United Kingdom
*{xli045, asdouglas}@ntu.edu.sg †jain7@llnl.gov ‡s.fahmy@warwick.ac.uk

***Abstract*—Coarse-grained overlays improve FPGA design productivity by providing fast compilation and software like programmability. Soft processor based overlays with well-defined ISAs are attractive to application developers due to their ease of use. However, these overlays have significant FPGA resource overheads. Time multiplexed (TM) CGRA-like overlays represent an interesting alternative as they are able to change their behavior on a cycle by cycle basis while the compute kernel executes. This reduces the FPGA resource needed, but at the cost of a higher initiation interval (II) and hence reduced throughput.**

The fully flexible routing network of current CGRA-like overlays results in high FPGA resource usage. However, many application kernels are acyclic and can be implemented using a much simpler linear feed-forward routing network. This paper examines a DSP block based TM overlay with linear interconnect where the overlay architecture takes account of the application kernels' characteristics and the underlying FPGA architecture, so as to minimize the II and the FPGA resource usage. We examine a number of architectural extensions to the DSP block based functional unit to improve the II, throughput and latency. The results show an average 70% reduction in II, with corresponding improvements in throughput and latency.

***Keywords*-Reconfigurable system, overlay architecture, FPGA**

I. Introduction

There has been a resurgence in FPGA-based accelerators due to developments in the cloud computing and IoT domains. FPGA accelerators are often custom designed to achieve maximum performance, using conventional RTL hardware design techniques, and as such, are only applied to specific algorithms in specific applications, and hence are fixed, negating many of the benefits of the programmable FPGA device. This design process is long and complex, requiring low-level device expertise and special knowledge of both hardware and software systems, resulting in major design productivity issues and long compilation times, which limit the mainstream adoption of FPGA based accelerators for general purpose computing.

High-level synthesis (HLS) has been widely adopted by EDA tool vendors to address the design productivity issue. However, to maximize performance, detailed low-level design effort is often still required, making design difficult for non-experts. Additionally, while HLS tools have allowed designers to focus on high-level functionality instead of low-level details, the back-end flow (specifically the FPGA place and route) requires very long compilation times, particularly for large designs, contributing to the lack of productivity and mainstream adoption of FPGAs. In many cases, the time required to change an FPGA configuration limits hardware accelerators to predesigned static (i.e. fixed) implementations, negating the fundamental benefit of FPGAs. To be more appealing to the broader group of application developers, who are used to software API abstractions and fast development cycles, the FPGA hardware resource needs to be better abstracted.

One possible solution to this problem is to use an overlay (a programmable coarse-grained hardware abstraction layer on top of the FPGA fabric) as this simplifies both the hardware design and mapping process. This then allows the FPGA to be treated as a virtualized execution platform that both abstracts the hardware details and enables runtime management support, so that the hardware can be viewed as just another software-managed task, possibly even controlled by the OS or a hypervisor [10]. This results in better application management, and has the potential for allowing portability across devices, software-like programmability by mapping from high-level descriptions, better design reuse, fast compilation by avoiding the complex FPGA design flow (particularly the very slow place and route process), resulting in improved design productivity. Another significant advantage is rapid application swapping (a hardware context switch) as coarse-grained overlay architectures have smaller configuration data sizes than fine-grained FPGAs. The major problem is that many of the current overlays are not efficient (in area, power, throughput, etc.) and still require FPGA-like configuration times, as in many cases the overlay needs to change when the application requiring acceleration needs to change.

II. Related Work

Overlays come in many forms, with the most common being spatially configured [9], [6] and time multiplexed (TM) [24], [16]. Spatially configured overlays fully unroll the kernel onto a pipelined array of FUs, resulting in an initiation interval (II) of 1. They provide high performance, but require significant FPGA resources. TM overlays change their behavior on a cycle by cycle basis, thus reducing the amount of FPGA resource dedicated to the functional unit (FU) and interconnect, but at the cost of a higher II and hence a reduced throughput.

Most successful TM overlays are based on soft processors. The more performance oriented ones include, SIMD

Octavo [13], VectorBlox MXP [24] and VLIW TILT [19]. A massively parallel overlay, called GRVI Phalanx [7], based on the RISC-V processor and the Hoplite NOC [11] mapped 1680 RISC-V cores onto an UltraScale+ VU9P. These overlays have the advantage of a well-known, well-designed ISA which makes them easy to use, however, they utilize a large amount of FPGA resource and have a significant power consumption.

An alternative solution is to build arrays of customized TM FUs and interconnect on the FPGA, similar to CGRAs [17]. A number of different interconnect styles for connecting between FUs can be used, with the most common being: island style [6], [8], nearest neighbor [20], [16] and to a lesser extent linear interconnect [3], [9]. The overhead of the interconnect network, particularly for island style and nearest neighbor interconnects, contribute to a significant FPGA resource utilization. Examples of CGRA-like TM overlays include:

CARBON [2], a CGRA-like overlay implemented as a 2×2 array of tiles on a Stratix III FPGA. Each CARBON tile has an FU with a programmable ALU and instruction memory, supporting up to 256 instructions. CARBON has a large FU resource requirement with a relatively slow speed which limits its scalability, compared to the overlays discussed below.

The SCGRA overlay [16] was proposed to address FPGA design productivity issues. Application specific SCGRA overlays were implemented on Zynq [15], achieving a speedup of up to 9× compared to the same application running on the Zynq ARM processor. The 250 MHz FU consists of an ALU, multiport data memory (256×32 bits) and customizable depth instruction ROM (Supporting 72-bit instructions) resulting in significant BRAM utilization. Fast application context switching is not possible as the full FPGA bitstream needs to be reconfigured for a compute kernel change.

The reMORPH overlay [20] better targets the FPGA fabric, with an FU consuming 1 DSP Block, 3 block RAMs, 196 LUTs and 41 registers. To reduce overhead, the reMORPH FU does not use decoders resulting in a 72-bit instruction memory (supporting up to 512 instructions) which also over utilizes the BRAMs. reMORPH uses a nearest neighbor style of non-programmable interconnect, which is adapted using partial reconfiguration at runtime, and hence, suffers from the same slow hardware context switch problem as SCGRA.

Many TM overlays have large area overheads due to the routing resources, or large instruction storage requirements. To address these problems, we propose a streaming architecture based on feed-forward pipelined datapaths, as streaming based accelerators have been highly successful when implemented in FPGAs [18], [23]. Targeting highly compute intensive algorithms with little control and relatively simple dependencies allows us to use a linear interconnect structure, where data flows in a single direction from one FU to the next, thus minimizing the interconnect requirements. This structure then enables the use of a very simple and efficient streaming memory interface. The instruction storage is also reduced, as the architecture allows us to store just those instructions used by an individual FU. The reduced instruction and control requirements means that a lightweight processor architecture can be used, similar to the DSP based iDEA processor [4], further reducing the hardware resource requirements while achieving a relatively high operating frequency.

III. Linear TM Overlay

While overlays with a general-purpose mesh-based interconnect allow for flexible communication between each FU, they introduce a significant resource overhead associated with the routing network. In many cases, a simple linear interconnect structure can be used instead. In a TM overlay, this reduces the highly flexible interconnect to a direct connection between FUs, as in Fig. 1, and allows data flow graph (DFG) nodes from the same scheduling time step to be allocated to individual FUs [14]. The linear overlay consists of a streaming data interface made up of Distributed RAM (DRAM) acting as a FIFO, which feeds the cascade of time-multiplexed FUs, with another DRAM-based FIFO at the output. Tasks are scheduled to the overlay using ASAP scheduling, with nodes at the same (horizontal) level allocated to a single FU. For example, Fig. 2a shows the medical imaging 'gradient' benchmark [5], while Fig. 2b shows the resulting DFG. This example requires 4 FU stages, where the first stage contains 4 subtract operations which would execute on the first FU, then the 4 multiplication operations execute on the second FU, and so on.

The FU uses the same principle as the iDEA DSP-based processor [4], and requires 1 DSP block, 160 LUTs and 293 FFs and runs at 325 MHz on a Xilinx Zynq XC7Z020. The FU consists of a LUTRAM-based instruction memory (IM) and register file (RF), and a DSP-based ALU, as shown in Fig. 3 (excluding all of the logic in the four dashed boxes).

The major advantage of TM overlays is that an application kernel can be mapped to fewer FUs, reducing resource consumption at the expense of II. The example of Fig. 2b can be mapped onto a linear overlay with 4 FUs using ASAP scheduling and has an II of 11, consisting of 5 cycles for data entry, 4 cycles for the 4 subtract operations, 1 cycle for data output and 1 cycle to flush the pipeline. By comparison, a spatially configured overlay would have an II of 1, requiring 11 FUs. However, using ASAP-based scheduling means that the overlay has a depth equal to the critical path of the DFG, and

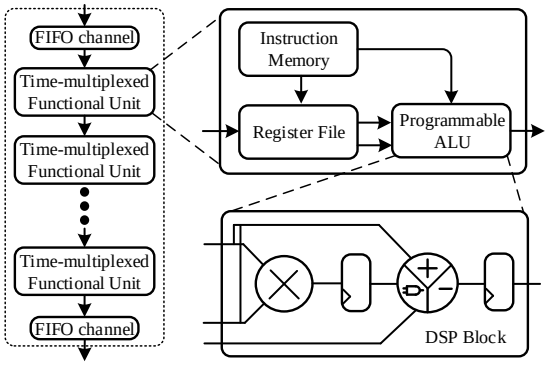
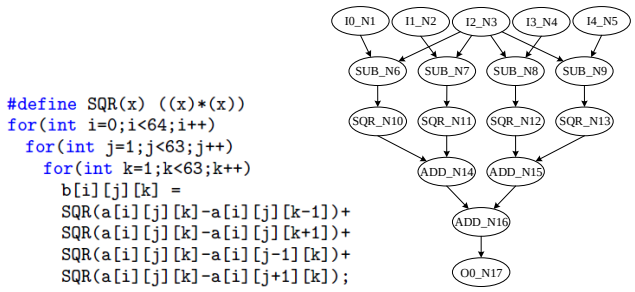


Fig. 1: A linear TM overlay.



(a) C Source Code (b) Data Flow Graph

Fig. 2: The 'gradient' benchmark.

must be re-sized for each new application kernel, thus limiting its usefulness. Whereas, a small linear overlay with a fixed depth that is able to map larger more general purpose compute kernels would be much more useful. In the next sections, we examine mechanisms to increase the throughput and usability.

A. Architectural Enhancements

The II is a critical metric for determining the throughput of an accelerator. The II of the overlay in [14] is obtained by the maximum of the number of data load operations plus the number of execution operations with 2 additional clock cycles needed to flush the pipeline among the FUs, as in Equation 1. This is especially large for DFGs with a large number of inputs and operation nodes in the first scheduling stage.

$$II = \max_{FU}\{\#load + \#op + 2\} \quad (1)$$

1) Rotating Register File: The most obvious way to reduce the II of Equation 1 is to overlap the loading of input data with instruction execution. Instead of adding additional complexity into the FU to support double-buffering, a rotating register file [22] is used to support the overlap of data written into the RF with subsequent instruction execution. The original design of [14] used a RAM32M primitive with a dual port configuration (1 read, 1 read/write), whereas the rotating RF version requires a quad port configuration to support 2 reads and 1 write. The new FU, shown in Fig. 3, includes the offset counter but not the four shaded registers to the left of the RAM32M RF block or the two shaded registers to the right of the DSP block. This design requires 1 DSP block, 196 LUTs, 237 FFs and has a frequency of 334 MHz on a Zynq XC7Z020 (610 MHz on a Virtex-7 VC707). We refer to this new design as version 1 (V1), and the II is determined as:

$$II_{V1} = \max_{FU}\{\#load + 1, \#op + 2\} \quad (2)$$

where the extra cycle in data load is to separate data blocks.

2) Replicating the Stream Datapath: The II can be reduced, at the expense of an increased data bandwidth requirement, by increasing parallelism. Replicating the data processing part of the FU (shown within the right dash-dot box) and increasing the data I/O to 64 bits doubles data throughput (halving the II). This design which reuses the instruction memory and other control circuitry of V1 is called version 2 (V2). It requires 2 DSP blocks, 292 LUTs and 333 FFs, operates at a frequency of 335 MHz and has an II half that of Equation 2.

TABLE I: Comparison of different FU designs.

	[14]	V1	V2	V3	V4	V5
DSPs	1	1	2	1	1	1
LUTs	160	196	292	212	207	248
FFs	293	237	333	228	163	126
Fmax	325	334	335	323	254	182
IWP	–	–	–	5	4	3

The resource consumption and maximum frequency for the various FU designs on a Zynq XC7Z020 are listed in Table I. The V1 FU consumes around 22% more LUTs than that of [14], mainly due to the addition of the RAM32M primitive and the offset counter. The resource consumption of V2 is less than twice that of V1, with a similar frequency to V1.

3) FU Write-back: The main disadvantage with these overlays is that they are feed-forward only, and thus the overlay depth (and the number of FUs) depends on the critical path of the DFG. If output data is written back to the RF, multiple nodes on the DFG critical path could be combined within the same scheduling stage, thus reducing the overlay depth. Without write-back, when the application kernel changes the overlay also needs to change, requiring overlay reconfiguration between kernels which significantly impacts the hardware context switch time. Thus, a fixed architecture which is able to handle a range of more general kernels would improve execution time when multiple kernels need to be accelerated.

Introducing data write-back is relatively simple and involves feeding the *Data_out* signal back into the FU and multiplexing it with the *Data_in* signal, as shown in the lower left dashed box in Fig. 3. This requires that the instruction format of [14] be modified with two extra bits added, a write-back (*WB*) bit and a no data forward (*NDF*) bit. Both bits are needed as there is a possibility that the output data will be written back to the RF and bypassed to the next FU stage. Rather than adding two extra bits to the (already) 32-bit instruction, we note that the DSP primitive is only used to support operations with 2 or 3 operands (which means the D port is unused and can be disabled). This means that three bits of the DSP *inmode* field can be hardwired, allowing the use of 1-bit as the *WB* flag, 1-bit as the *NDF* flag, with 1-bit reserved for future use. The *Valid_in* signal and the delayed *WB* flag are then used to select between the two different data sources in the write-back logic.

Table I shows the resource utilization, operating frequency and internal write-back path (IWP) for three different implementations of the FU with write-back, referred to as V3, V4 and V5. The V3 FU is identical to V1, except that the write-back logic is added. That is, it includes all circuitry in Fig. 3 apart from the left and right shaded registers. The IWP is five, comprising one cycle in the RF, one at the register between the RF and the input map logic, and three in the DSP block. This overlay operates at a frequency close to that of the non-WB overlays. To reduce the IWP, the registers between the RF

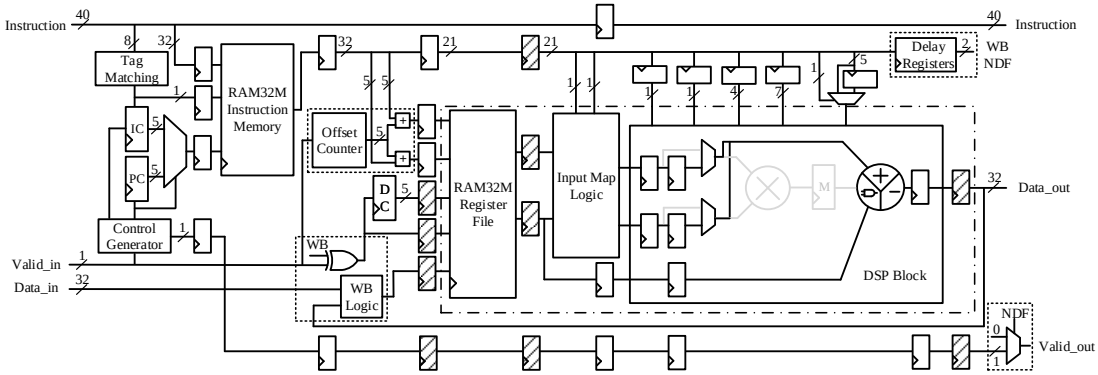


Fig. 3: Proposed time-multiplexed functional unit.

RAM32M primitive and the input map logic can be deleted, resulting in a slight frequency reduction. This FU, referred to as V4, is identical to V3 except that all shaded registers in Fig. 3 are removed. It has an IWP of 4 and a frequency of 254MHz. A further reduction in the IWP can be achieved by reducing the pipeline depth of the DSP block from three to two, resulting in an IWP of 3 and a frequency of 182MHz.

The V3-V5 FUs can then be implemented as a fixed depth overlay, as in Fig. 1. We propose implementing two depth 8 overlays in a single tile, with replicated tiles connected via a lightweight NOC, such as in [11]. The two overlays in a tile could either be connected in series (to form a single depth 16 overlay) or connected in parallel to produce a depth 8 overlay with dual datapaths, similar to the V2 based overlay.

As data forwarding within a DSP block is not possible, due to the inability to access internal signals, it is important to understand the impact of a fixed depth overlay when write-back is used. When scheduling DFG nodes to the overlay, any dependency between nodes will require the insertion of NOPs, equal to the IWP, unless other non-dependant nodes can be scheduled between the nodes with the dependency.

IV. Compiling to the Overlay

There are two separate design processes for mapping an application to an overlay. The first is the overlay implementation which is carried out offline using the conventional FPGA design flow. At power-on, the bitstream (consisting of the overlay, memory and communication interfaces, and any other components) is used to configure the FPGA. The second involves mapping the application kernel to the overlay. To allow for fast vendor independent mapping to the overlay we developed our own mapping tool flow. This involves, DFG extraction from high-level compute kernels, scheduling the DFG nodes onto the overlay, and finally, instruction generation for each FU. This is typically done offline, however it could also be performed as part of a just-in-time mapping strategy. On Zynq, the ARM processor loads the kernel configuration into the overlay pipeline and initiates kernel execution. Our mapping flow is described below using the previous example.

Kernel Mapping: The open source HercuLeS HLS tool [12] is used to transform a 'C' description of the compute kernel to a DFG description, where nodes represent operations and edges represent data flow between operations, as shown in Fig. 2b. For the V1 and V2 based overlays, ASAP scheduling is used which results in no data dependencies between operations at the same scheduling stage, as in [14], with nodes in each scheduling stage then being allocated to a single V1 or V2 FU for execution. The set of instructions from the sequenced DFG is identified, then the cycle-by-cycle execution pattern is formed which interleaves load/store and arithmetic/ALU operations, as shown in Table II. For the 'gradient' benchmark, the II is reduced from 11 (in [14]) to 6 (V1) or 3 (V2) with the same ASAP scheduling. This translates to a throughput of 0.59 Giga-operations/s (GOPS) for the V1 based overlay with a latency of 86.8 ns (1.11 GOPS and 92.4 ns for V2). Lastly the 32-bit FU instructions are generated.

Typically, most of the existing CGRA architectures adopt Modulo scheduling [21], or a derivative algorithm, to achieve a minimum II. However, Modulo scheduling is based on the assumption that each operation node is executed in 1 cycle and the transfer of data between two arbitrary FUs completes in 1 cycle, which is not realistic for highly pipelined architectures. Instead, for a fixed depth overlay we use an iterative greedy scheduling strategy which groups DFG nodes at each scheduling step into clusters and then adds DFG

TABLE II: First 32 cycles of the 'gradient' schedule (II=6).

cycle	FU0		FU1		FU2		FU4	
1	Load R0							
2	Load R1							
3	Load R2							
4	Load R3							
5	Load R4							
6		SUB (R0 R2)						
7	Load R0	SUB (R1 R2)						
8	Load R1	SUB (R2 R3)						
9	Load R2	SUB (R2 R4)	Load R0					
10	Load R3		Load R1					
11	Load R4		Load R2					
12		SUB (R0 R2)	Load R3					
13	Load R0	SUB (R1 R2)		SQR (R0 R0)				
14	Load R1	SUB (R2 R3)		SQR (R1 R1)				
15	Load R2	SUB (R2 R4)	Load R0	SQR (R2 R2)				
16	Load R3		Load R1	SQR (R3 R3)	Load R0			
17	Load R4		Load R2		Load R1			
18		SUB (R0 R2)	Load R3		Load R2			
19	Load R0	SUB (R1 R2)		SQR (R0 R0)	Load R3			
20	Load R1	SUB (R2 R3)		SQR (R1 R1)		ADD (R0 R1)		
21	Load R2	SUB (R2 R4)	Load R0	SQR (R2 R2)		ADD (R2 R3)		
22	Load R3		Load R1	SQR (R3 R3)	Load R0			
23	Load R4		Load R2		Load R1		Load R0	
24		SUB (R0 R2)	Load R3		Load R2		Load R1	
25	Load R0	SUB (R1 R2)		SQR (R0 R0)	Load R3			ADD (R0 R1)
26	Load R1	SUB (R2 R3)		SQR (R1 R1)		ADD (R0 R1)		
27	Load R2	SUB (R2 R4)	Load R0	SQR (R2 R2)		ADD (R2 R3)		
28	Load R3		Load R1	SQR (R3 R3)	Load R0			
29	Load R4		Load R2		Load R1		Load R0	
30		SUB (R0 R2)	Load R3		Load R2		Load R1	
31	Load R0	SUB (R1 R2)		SQR (R0 R0)	Load R3			ADD (R0 R1)
32	Load R1	SUB (R2 R3)		SQR (R1 R1)		ADD (R0 R1)		

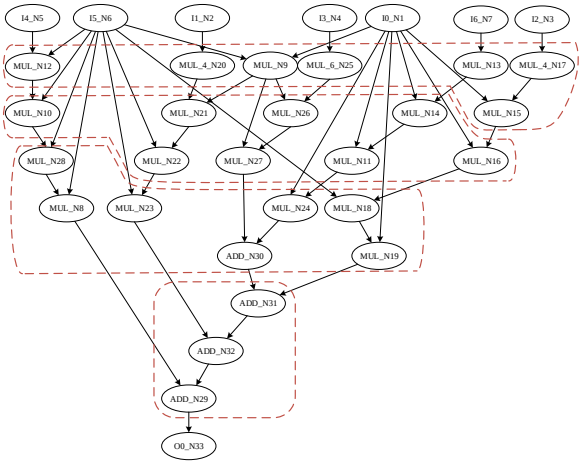


Fig. 4: Data flow graph of the 'qspline' benchmark.

nodes along the critical path from subsequent clusters, while balancing the II across all clusters. The number of scheduling clusters is equal to the overlay depth. Due to space constraints, the scheduling algorithm will not be discussed further.

As an example of fixed depth overlay scheduling, consider the 'qspline' benchmark, of Fig. 4. Here, the critical path is 8 and we map to a depth 4 overlay (4 FUs). Scheduling produces the 4 instruction clusters shown in Fig. 4 (using red dashes). NOPs (equal to IWP-1) must be added between dependant instructions (DFG nodes) unless other non-dependant instructions can be scheduled in between. For example, in the first (top) cluster, Node 17 is scheduled, followed by 13, 25, 9, 20, and 12, before 15 is scheduled. Hence, the dependency between 17 and 15 is resolved and no NOPs are inserted. Similarly for the 2nd cluster, scheduling as: 14, 26, 21, 10, 16, 11, 27, 22, resolves dependencies 14-11, 26-27, and 21-22, for all overlay versions. In cluster three, scheduling as: 18, 24, 28, 23, 19, 30, 8, resolves all dependencies for the V4 and V5 overlays, but not for the V3 overlay, which with an IWP of 5 requires 4 operations between dependant nodes. Hence, a single NOP must be added between 23 and 19 which then resolves all 4 sets of dependant instructions. For the 4th cluster, graph balancing is performed, and the two additions scheduled, followed by IWP-1 NOPs before the final addition.

The consequences of a fixed depth overlay are an increase in the II with a corresponding reduction in the throughput, but with a significant reduction in the latency. For the 'qspline' benchmark, the V3 overlay has an II of 15, with a throughput of 0.51 GOPS and a latency of 125 ns, while the V4 overlay has an II of 14, a throughput of 0.43 GOPS and a latency of 148 ns. This compares to the depth 8 V1 overlay with an II of 11, a throughput of 0.69 GOPS and a latency of 234ns.

V. Experimental Evaluation

We compare the performance of our linear TM overlays using a set of compute kernels from [8], [1], as shown in Table III. The V1 (1 DSP, no WB), V2 (2 DSP, no WB), V3 (WB, IWP=5) and V4 (WB, IWP=4) overlays are compared

TABLE III: DFG characteristics of benchmark set.

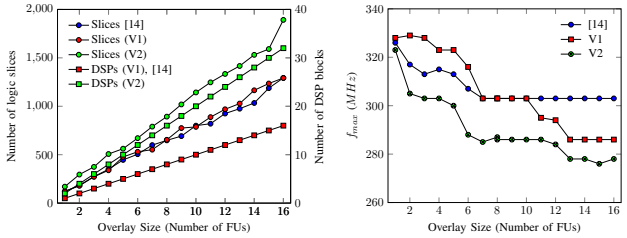
No.	Benchmark	I/O	#Ops	Depth	$II_{[14]}$	II_{V1}	II_{V2}	II_{V3}	II_{V4}
1.	chebyshev	1/1	7	7	6	4	2	4	4
2.	mibench	3/1	13	6	14	8	4	8	8
3.	qspline	7/1	25	8	19	11	5.5	11	11
4.	sgfilter	2/1	18	9	13	8	4	8	8
5.	poly5	3/1	27	9	19	11	5.5	11	11
6.	poly6	3/1	44	11	25	14	7	13	12
7.	poly7	3/1	39	13	24	14	7	20	17
8.	poly8	3/1	32	11	21	12	6	16	14

to the overlay in [14]. V1, V2 and the overlay in [14] have a depth equal to the critical path, and are configured on a kernel by kernel basis, while V3 and V4 have a fixed depth of eight. All overlays are implemented on a Zynq XC7Z020.

The FPGA DSP and logic slice utilization, and operating frequency, for different depth V1 and V2 overlays are shown in Fig. 5 (V3 and V4 are not included in this figure as they have a fixed depth). A depth 8 V1 overlay consumes 654 logic slices and 8 DSP slices which represents less than 5% of the logic and DSP resources on Zynq. The depth 8 V2 overlay consumes 893 logic slices and 16 DSP blocks or less than 8% of the Zynq resources. By comparison, the fixed depth (of 8) V3 (and V4) overlay consumes 814 (817) logic slices, 8 (8) DSP slices and operates at a frequency of 286MHz (233MHz).

The DFG characteristics (number of I/O, number of arithmetic operations and graph depth) for the chosen benchmarks and the II achieved when mapped to the various overlays are shown in Table III. For the first three benchmarks, which have a depth ≤ 8, ASAP scheduling is used to map to the V3 and V4 overlays, and thus, the II is the same as for the V1 overlay. The V1 (V2) overlay has an average 42% (71%) reduction in the II, compared to [14]. The V3 (V4) overlay has an average 34% (40%) reduction in the II for the depth > 8 benchmarks.

Fig. 6 shows the throughput and latency of the different overlays for the benchmarks given in Table III. In terms of throughput, all overlays have a higher throughput than the overlay of [14]. This is because interleaving data transfer with execution reduces the II and hence improves throughput. The two DSP V2 overlay has approximately twice the throughput as the V1 overlay, but also requires twice the data bandwidth. The size of both of these overlays is dependant on the depth (critical path) of the application kernel's DFG, and needs to be reconfigured when the application kernel changes. A depth 8



(a) Resource Usage (b) f_{max} Drop

Fig. 5: V1 and V2 overlay scalability on Zynq XC7Z020.

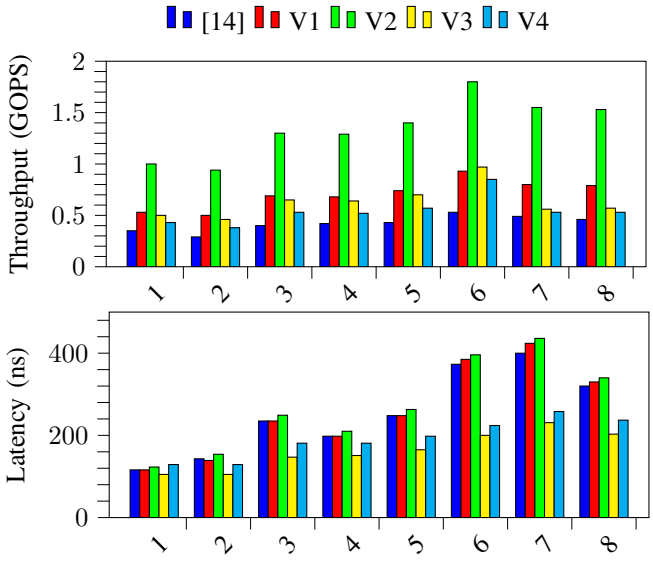


Fig. 6: Throughput and latency for the benchmarks.

V1 (V2) overlay requires a minimum reconfigurable region of 7 (9) CLB tiles and 1 (2) DSP tile with a configuration time of 0.73 (1.02) ms using the processor configuration access port (PCAP). Additionally, the overlays require a further 0.29μs to load the configuration data for the largest benchmark.

The single DSP V3 overlay has a throughput similar to the V1 overlay, with an average reduction of just 10%. The V4 overlay has a slightly reduced throughput as it operates at a lower frequency due to the removal of pipeline registers to reduce the IWP. The V3 and V4 overlays both have a fixed depth (in these experiments a depth of 8 is used). Adding write-back capabilities allows larger kernels to be mapped to a smaller number of FUs, removing the requirement that the overlay depth must be the same as the kernel critical path. This eliminates the need to reconfigure the overlay when the application kernel changes, making the overlay more general purpose (but requiring a different scheduling strategy). Thus, a hardware context switch on the V3 overlay requires just 0.25μs for the largest benchmark, representing a 2900$\times$ reduction compared to the V1 overlay.

The latency is heavily dependent on the depth of the overlay. For the V1 and V2 overlays and the overlay of [14], the overlay depth is equal to the DFG depth, due to the ASAP scheduling strategy used, and hence these overlays all have a larger latency. The V3 and V4 overlays generally show a significant reduction in the latency, particularly for larger depth DFGs, due to the fixed overlay depth.

VI. Conclusion

We have presented an area efficient FPGA overlay with a linear connection of TM FUs based on the Xilinx DSP48E1. Interleaving data transfer with instruction execution significantly reduces the II with a resulting increase in throughput. Introducing write-back into the FU design allows the overlay depth to be fixed. This eliminates the need to reconfigure the overlay if the application kernel changes, making the overlay more general. These changes significantly reduce the latency with just a small decrease in throughput. The V3 overlay has the best performance, as for a fixed data bandwidth, it has comparable throughput with a significantly reduced latency.

References

[1] D. Bini and B. Mourrain. Polynomial test suite, 1996. *See http://www-sop. inria. fr/saga/POL*.

[2] A. D. Brant. *Coarse and fine grain programmable overlay architectures for FPGAs*. PhD thesis, University of British Columbia, 2013.

[3] D. Capalija and T. S. Abdelrahman. Towards synthesis-free JIT compilation to commodity FPGAs. In *FCCM'11*, pages 202–205, 2011.

[4] H. Y. Cheah, F. Brosser, S. A. Fahmy, and D. L. Maskell. The iDEA DSP block-based soft processor for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(3):19:119:23, 2014.

[5] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou. A fully pipelined and dynamically composable architecture of CGRA. In *FCCM'14*, pages 9–16, 2014.

[6] J. Coole and G. Stitt. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *CODES+ISSS'10*, pages 13–22, 2010.

[7] J. Gray. Grvi phalanx: A massively parallel RISC-V FPGA accelerator accelerator. In *FCCM'16*, pages 17–20, 2016.

[8] A. K. Jain, S. A. Fahmy, and D. L. Maskell. Efficient overlay architecture based on DSP blocks. In *FCCM'15*, pages 25–28, 2015.

[9] A. K. Jain, X. Li, P. Singhai, D. L. Maskell, and S. A. Fahmy. Deco: A DSP block based FPGA accelerator overlay with low overhead interconnect. In *FCCM'16*, pages 1–8, 2016.

[10] A. K. Jain, K. D. Pham, J. Cui, S. A. Fahmy, and D. L. Maskell. Virtualized execution and management of hardware tasks on a hybrid ARM-FPGA platform. *J. Signal Process. Syst.*, 77(1-2):61–76, 2014.

[11] N. Kapre and J. Gray. Hoplite: Building austere overlay NoCs for FPGAs. In *FPL'15*, pages 1–8, 2015.

[12] N. Kavvadias and K. Masselos. Hardware design space exploration using HercuLeS HLS. In *PCI'13*, pages 195–202, 2013.

[13] C. E. Laforest and J. H. Anderson. Microarchitectural comparison of the MXP and Octavo soft-processor FPGA overlays. *ACM TRETS*, 10(3):19, 2017.

[14] X. Li, A. Jain, D. Maskell, and S. A. Fahmy. An area-efficient FPGA overlay using DSP block based time-multiplexed functional units. In *OLAF'16*, 2016.

[15] C. Liu, H.-C. Ng, and H. K.-H. So. QuickDough: a rapid FPGA loop accelerator design framework using soft CGRA overlay. In *FPT'15*, pages 56–63, 2015.

[16] C. Liu, C. L. Yu, and H. K.-H. So. A soft coarse-grained reconfigurable array based high-level synthesis methodology: Promoting design productivity and exploring extreme FPGA frequency. In *FCCM'13*, pages 228–228, 2013.

[17] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *FPL'03*, pages 61–70. 2003.

[18] T. F. Oliver, B. Schmidt, and D. L. Maskell. Reconfigurable architectures for bio-sequence database scanning on FPGAs. *IEEE Trans. Circuits Syst. II, Exp. Briefs*, 52(12):851–855, 2005.

[19] K. Ovtcharov, I. Tili, and J. G. Steffan. TILT: a multithreaded VLIW soft processor family. In *FPL'13*, pages 1–4, 2013.

[20] K. Paul, C. Dash, and M. S. Moghaddam. reMORPH: a runtime reconfigurable architecture. In *DSD'12*, pages 26–33, 2012.

[21] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *MICRO'94*, pages 63–74, 1994.

[22] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. *ACM SIGPLAN Notices*, 27(7):283–299, 1992.

[23] F. Saqib, A. Dutta, J. Plusquellic, P. Ortiz, and M. S. Pattichis. Pipelined decision tree classification accelerator implementation in FPGA (DT-CAIF). *IEEE Trans. Comput*, 64(1):280–285, 2015.

[24] A. Severance and G. G. Lemieux. Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor. In *CODES+ISSS'13*, pages 1–10. IEEE, 2013.