# Are Coarse-Grained Overlays Ready for General Purpose Application Acceleration on FPGAs?

Abhishek Kumar Jain*, Douglas L. Maskell* and Suhaib A. Fahmy†
*School of Computer Engineering, Nanyang Technological University, Singapore
†School of Engineering, University of Warwick, United Kingdom
Email: {abhishek013, asdouglas}@ntu.edu.sg, s.fahmy@warwick.ac.uk

***Abstract*—Combining processors with hardware accelerators has become a norm with systems-on-chip (SoCs) ever present in modern compute devices. Heterogeneous programmable system on chip platforms sometimes referred to as hybrid FPGAs, tightly couple general purpose processors with high performance reconfigurable fabrics, providing a more flexible alternative. We can now think of a software application with hardware accelerated portions that are reconfigured at runtime. While such ideas have been explored in the past, modern hybrid FPGAs are the first commercial platforms to enable this move to a more software oriented view, where reconfiguration enables hardware resources to be shared by multiple tasks in a bigger application. However, while the rapidly increasing logic density and more capable hard resources found in modern hybrid FPGA devices should make them widely deployable, they remain constrained within specialist application domains. This is due to both design productivity issues and a lack of suitable hardware abstraction to eliminate the need for working with platform-specific details, as server and desktop virtualization has done in a more general sense. To allow mainstream adoption of FPGA based accelerators in general purpose computing, there is a need to virtualize FPGAs and make them more accessible to application developers who are accustomed to software API abstractions and fast development cycles. In this paper, we discuss the role of overlay architectures in enabling general purpose FPGA application acceleration.**

## I. Introduction

The internet of things has resulted in a diverse range of computing requirements, but with one central focus, the need for low power computing [1]. While much of this can be handled by low power processors, possibly with cloud offloading, more compute intensive applications will require a different approach. High performance processors and/or graphics processors (GPUs) are unable to meet the ever increasing demand for computing power within the tight power budget required by high performance embedded systems. Heterogeneous programmable systems on chip (PSoC) platforms, which tightly couple general purpose processor(s) (GPPs) with a high performance reconfigurable FPGA fabric [2], provide a more flexible alternative to conventional system on chip (SoC) architectures while providing better power and performance characteristics than high performance CPUs and GPUs.

While this reconfigurable computing approach has been discussed before [3], [4], the more capable multicore processors in these newer devices provides the ability to move the focus of reconfigurable computing systems away from static (or quasi-static) accelerators to a more software oriented view, where reconfiguration is a key enabler for reusing available hardware resources among multiple tasks. However, while the rapidly increasing logic density and more capable hard resources found in modern PSoC platforms, should make them applicable to a wider range of domains, such platforms have not seen significant use beyond specialist application domains, such as digital signal processing and communications. Traditional approaches to managing execution and scheduling of hardware tasks are inappropriate and cumbersome for exploiting these platforms. In addition, the design process is complex, requiring low-level device expertise and specialist knowledge of both hardware and software systems, resulting in major design productivity issues.

One technique for addressing design productivity is to use high-level synthesis (HLS) [5]. Advancements in HLS tools have helped raise the level of programming abstraction from the low register-transfer-level (RTL) used in hardware description languages (such as Verilog) to high level languages, such as C or C++. However, achieving the desired performance often still requires detailed low-level design engineering effort that is difficult for non-experts. Additionally, even though HLS tools have improved in efficiency, allowing designers to focus on high level functionality instead of low-level implementation details, the prohibitive compilation time (specifically the place and route times in the backend flow) is a major impediment which still limits productivity and mainstream adoption [6].

Another major stumbling block is the lack of a suitable abstraction at the hardware computing level, to eliminate the reliance on platform-specific detail, as has been achieved with server and desktop virtualization. A key example of virtualization in a modern paradigm is cloud computing, where virtual resources are available on demand, with runtime mapping to physical systems abstracted from the user. So far, virtualization has focused primarily on conventional processor-based computing systems where high level management of computing tasks is supported by having a number of abstraction layers at different abstraction levels. However there is no agreed upon abstraction for FPGA fabrics.

To allow mainstream adoption of FPGA based accelerators in general purpose computing and virtualized execution of software and hardware tasks, there is a growing need to virtualize FPGAs and make them more accessible to application developers who are accustomed to software API abstractions and fast development cycles. The lack of platform abstraction and

application portability prevents design reuse and adoption of these platforms for mainstream computing. Hence we require a revised look at how to effectively exploit the key advantages of reconfigurable hardware while abstracting implementation details within a software-centric processor-based system.

One possible solution is to treat the execution and management of software and hardware tasks in the same way, using a hypervisor or operating system (OS) such that the hardware fabric is viewed as just another software-managed task [7], [8]. This enables more shared use, while ensuring better isolation and predictability. This run-time management, including FPGA configuration and interprocess data communication, was recently demonstrated using both a hypervisor [9] and within the Linux OS [10]. The use of a programmable coarse-grained hardware abstraction layer, on top of the FPGA, resulted in better application portability across devices, better design reuse, and rapid reconfiguration that is orders of magnitude faster than other reconfiguration approaches on FPGAs. This hardware abstraction layer is referred to as an overlay as it sits on top of the FPGA fabric allowing the user to program different functionality to it.

In this paper, we discuss an execution platform based on a virtual overlay sitting on top of the physical FPGA fabric of a commercial hybrid FPGA that not only abstracts the reconfigurable hardware details, such as the logic, memory, and I/O interfaces and their placement, but also provides runtime management support in order to facilitate virtualized execution of software and hardware tasks. This enables small, often used, sections of code to be mapped to dedicated hardware accelerators on demand. We show that the current state-of-the-art in FPGA overlays provides three orders of magnitude improvement in the hardware mapping process and the time required to switch between different hardware accelerators (a hardware context switch) compared to that of conventional approaches, while only suffering from a single order of magnitude reduction in area efficiency.

The remainder of the paper is organized as follows: Section II details the motivation for using overlays by providing a basic understanding of FPGA architecture and the key barriers to mainstream usage. Section III introduces the concept of coarse-grained overlays followed by a description of the two main types of overlay. Section IV provides a comparison of some of the more recent overlays from the research literature, as well as a comparison to a conventional FPGA-based hardware implementation. Section V examines the use of an FPGA overlay for general purpose application acceleration within a hybrid FPGA. Finally, we conclude in Section VI.

## II. Barriers to Mainstream Use of FPGAs

To better understand why FPGA devices have not achieved mainstream adoption among the wider computing community, we must first understand how FPGAs differ from alternative solutions, specifically traditional GPPs. The most fundamental difference relates to how an application is mapped to the these platforms. A GPP provides functionality to execute a compute kernel as a list of sequential instructions, whereas an FPGA architecture implements compute kernels by mapping them to fine grained resources, such as configurable logic blocks, and medium grained hard DSP blocks, Block RAMs, etc. These resources are interconnected via a fine-grained programmable island-style routing network to create a specialized datapath which implements the compute kernel. By exploiting parallelism in the algorithm, significant performance gains are possible.

### A. Low Level Hardware Design

FPGA accelerators are normally designed at a low level of abstraction (typically RTL) in order to obtain an efficient implementation, and this can consume more time and make reuse difficult when compared to a similar software design. To build an FPGA accelerator, designers typically start by manually converting the compute kernel into an fully pipelined datapath, specified using a hardware description language (HDL) such as Verilog or VHDL. The designer must specify the detailed structure of the datapath and must also define control for reading inputs from memories into buffers, stalling the datapath when buffers are full or empty, writing outputs to memory, and so on. For a typical FPGA device, a fully pipelined datapath implementing just several lines of C code may require 2-3 orders of magnitude more lines of HDL code, but results in significantly better performance by pipelining and exploiting parallelism. However this performance comes at the cost of significant design effort.

Additionally, a design for a reconfigurable device does not necessarily port well to the next hardware generation, making reconfigurable systems more difficult to work with. The designer must make a number of decisions, such as how to best fit the application to the device, including the datapath structure and the amount of parallelism. Applications are normally optimized for a specific target device, and are unable to execute on a smaller device or cannot take full advantage of the additional resources on a larger device.

Once the designer has a working design it must be implemented on the FPGA. The FPGA tool flow typically takes an RTL description of the design and first performs technology mapping to convert it into the fine-grained device resources, followed by placement and routing (PAR). Due to the fine granularity of the FPGA resources, the this process is complex and for large designs results in very lengthy place and route times.

### B. Reconfiguration Latency

The FPGA fabric, being programmable, is able to adapt to changing processing requirements, thus better utilising FPGA resources, while providing a more software centric approach to hardware design. This allows software applications to be profiled and partitioned, with the resulting hardware accelerator running on the FPGA fabric and the remaining software running on the GPP, with significant performance improvements. These accelerators can also be rapidly reconfigured by utilizing the ability to partially and dynamically reconfigure the functionality of the FPGA fabric. However, despite the

popularity and inherent capability of FPGAs for partial reconfiguration, whereby the FPGA operation is dynamically adapted to changing application requirements, this feature is not well supported by FPGA vendors and is hampered by slow reconfiguration times, poor CAD tool support, and large configuration file sizes. These issues make dynamic reconfiguration difficult and inefficient, resulting in most FPGAs being used with just a single configuration.

Initial implementations of dynamic reconfiguration [11], [12] required the reconfiguration of the whole hardware fabric. This resulted in significant configuration overhead, which severely limited its usefulness. Xilinx introduced the concept of dynamic partial reconfiguration (DPR) which reduced configuration time by allowing a smaller region of the fabric to be dynamically reconfigured at runtime. The concept of DPR on FPGA is one way of virtualizing hardware to allow implementiation of applications that are larger than the FPGA. DPR significantly improved reconfiguration performance [13], however the efficiency of the traditional design approach for DPR is heavily impacted by how a design is partitioned and floorplanned [14], tasks that require FPGA expertise. Furthermore, the commonly used configuration mechanism is highly sub-optimal in terms of throughput [15]. Despite numerous efforts in reducing reconfiguration times and improving CAD tool support for dynamic reconfiguration of FPGA fabric [16], [13], the implementation of rapidly reconfigurable hardware accelerators is still difficult.

### C. Coarse-Grained Reconfigurable Devices

The complexity in the FPGA tool flow due to the use of fine-grained FPGA resources can be easily demonstrated by example. Fig. 1(a) shows the placed and routed design of a simple 4 input 16-bit adder. Here, the FPGA design tools divide the design into basic circuit elements and map them to the fine-grained configurable logic blocks (CLBs). On the other hand, Fig 1(b) shows the placed and routed design of the same application on a coarse-grained architecture where compute blocks (or functional units (FU)) and interconnect have a 16-bit width, compared to single-bit tracks in the fine-grained FPGA implementation. It is clear that the PAR complexity is significantly reduced by using coarse-grained architectures, thus reducing compilation time. Another benefit of using a coarse-grained architecture is the reduced configuration data size and hence reduced reconfiguration latency which can allow faster context switching.

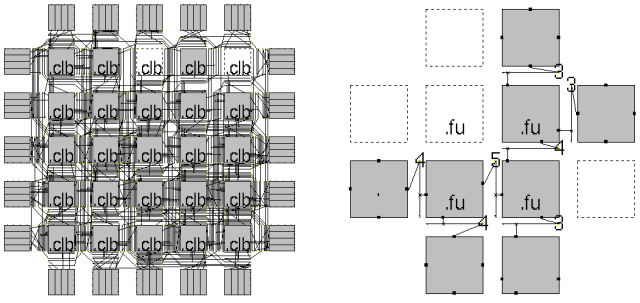


Fig. 1: Placement of Routing on (a) Fine-grained (b) Coarse-grained architecture.

Because of this apparent advantage, researchers have explored a number of ASIC implementations of coarse-grained reconfigurable architectures (CGRAs) [17], [18], [19], [20], [21], [22], [23], [24], [25]. Some key features that enabled these architectures to address signal processing and high performance computing problems more efficiently include: energy efficiency, ease of programming, fast compilation and reconfiguration. The Rapid [19] architecture was designed to implement computation-intensive and highly regular systolic streaming applications using an array of computing cells, each consisting of a multiplier, two ALUs, six general purpose registers, and three small local memories. Morphosys[20] was proposed as a coarse-grained, integrated reconfigurable SoC targeting high throughput applications such as multimedia and image processing. It consisted of a Tiny RISC processor core, an $8\times8$ reconfigurable array, context memory, frame buffer and a DMA controller. The REMARC CGRA was also proposed for multimedia applications and consisted of a MIPS ISA based core and an $8\times8$ reconfigurable logic array [26]. Each processing element of the array consists of a 16-bit processor, with processor execution controlled by instructions stored in a small local instruction memory.

The key attraction of coarse-grained reconfigurable devices is their near ASIC-like computational and energy efficiency and software-like engineering efficiency. At least for commercial products, the main market has been as a component in SoCs for efficiently implementing a specific range of DSP functions as part of a larger system. CGRAs have not been successfully developed as stand-alone systems that designers can incorporate at the board level because functional units are often too application specific to be efficient and useful for a wide range of applications. ASIC implementations of coarse-grain architectures also suffer from the design-time freeze of functional units and interconnect capabilities. It is hard to find a particular configuration that suits a wide enough set of applications for the approach to be viable as a stand-alone product. Hence there is a need for a mechanism where capabilities can be tailored to applications or adapted at runtime based on application needs.

## III. Coarse-Grained Overlay Architectures

One solution that has been explored extensively by researchers is to implement a coarse-grained reconfigurable architecture on top of a commercial FPGA device, referred to as a coarse-grained overlay. This allows the coarse-grained elements and structure, specifically the functional units (FUs) and interconnect to be modified at runtime according to application requirements. Compared to traditional FPGA design, a coarse-grained overlay architecture has several potential advantages. These include: improved designer productivity, better design portability, software-like programmability, faster application switching and enhanced security. This is motivated

by the fact that programs can be written at a higher level of abstraction with compilation to the overlay being several orders of magnitude faster than for the fine grained FPGA on which the overlay is implemented. That is, instead of the requirement for a full cycle through the FPGA vendor tools, overlay architectures present a simpler problem, that of programming an interconnected array of FUs. However, overlays are not intended to replace HLS tools and vendor-implementation tools and are instead intended to support FPGA usage models where programmability, abstraction, resource sharing, fast compilation, and design productivity are critical issues.

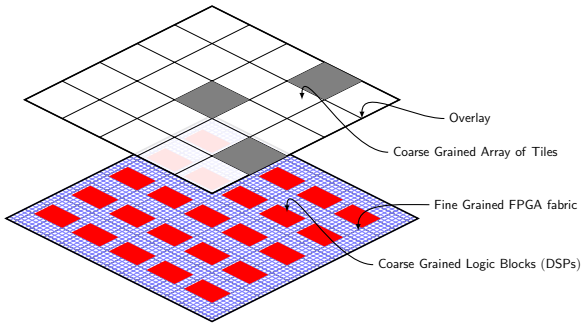


Fig. 2: Coarse-grained Overlay Architecture.

Overlays can be categorized based on their architecture, using the classification in [27], where 4 categories are defined: spatially configured, time multiplexed, packet switched, and circuit switched. Though examples of packet switched networks [27] and circuit switched networks exist, they are generally very resource hungry, and are unsuitable for large FPGA-based overlay architectures. As such, the majority of overlays are restricted to just two classes: spatially configured (SC) overlays; and time-multiplexed (TM) overlays, with both the FU and interconnect falling within one of these two categories.

### A. Time-Multiplexed Overlays

In a TM overlay, the compute and interconnect logic of the overlay change on a cycle by cycle basis while the compute kernel being executed [28]. This time-multiplexing of overlay resources among kernel operations eliminates the large FPGA resource overhead associated with SC overlays, but results in an initiation interval (II) between input data greater than one, resulting in reduced throughput. To achieve the best performance, the architecture must be carefully analyzed taking into account the characteristics of the application kernels and the underlying FPGA architecture.

Time multiplexed overlays normally have an instruction memory within each FU, with each FU behaving like a conventional processor core that is then time-multiplexed among multiple operations. Individual FUs are arranged into a (typically) two dimensional array, interconnected via a programmable interconnect. The interconnect structure is typically connects nearest neighbors (NN), allowing FUs to communicate only with neighboring FUs. The FUs themselves can be simple soft processors, such as the Xilinx Microblaze [29] or iDEA [30]. However, the movement of data and keeping the processors busy with computation presents a complex scheduling problem, resulting in poor performance.

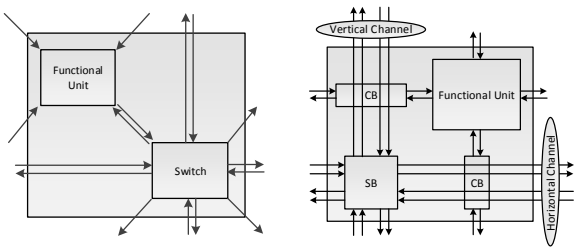


Fig. 3: Spatially-configured Overlay Tile Architectures: (a) DySER, and (b) DSP based.

### B. Spatially-Configured Overlays

The largest group of the coarse grained overlays in the research literature consist of SC FUs and SC interconnect networks [31], [32], [33], [34], which we refer to as an SCFU-SCN overlay. In an SCFU-SCN overlay, an FU executes a single arithmetic operation and data is transferred over a dedicated point-to-point links between FUs. That is, both the FU and the interconnect are unchanged while a compute kernel is executing, thus supporting maximum throughput by dedicating an individual FU to each kernel operation. This results in a fully pipelined, throughput oriented programmable datapath executing one kernel iteration per clock cycle, thus having an II of one. A number of different spatially configured interconnect strategies have been proposed, with the most common being: island style [31], [34], nearest neighbor (NN) [32], and to a lesser extent linear interconnect [35], [36]. However, many island style and nearest neighbor connected overlays suffer from high area overheads due to the resources required for the interconnect network and are unsuitable for large compute kernels due to the limited size of the overlay that can be mapped onto the FPGA fabric.

SC overlays have a number of advantages over time multiplexed overlays, such as the ability to exploit larger FPGAs to deliver scalable performance for data-parallel and throughput oriented applications. They are able to maintain extremely high throughput by employing deep pipelining within the architecture, as well as having drastically reduced compilation times and configuration data sizes due to the requirement for just one instruction per functional unit. But this flexibility comes at a cost in terms of area and performance overheads. Hence, a significant amount of research effort has recently been aimed at reducing area overheads and improving performance. The primary metrics considered include: the frequency and peak throughput of the overlay [32], programmability cost [32], peak throughput per unit area, the configuration data size, and configuration time [31]. With these in mind, we now discuss the key features, performance metrics, and overheads for a number of spatially configured overlay architectures proposed in the literature.

*1) Intermediate Fabrics:* An overlay architecture, referred to as an intermediate fabric (IF) [37], was proposed to support near-instantaneous placement and routing. A generic IF (consisting of 192 heterogeneous FUs with an island-style interconnect) was implemented on an Altera Stratix III FPGA in order to support fully parallel, pipelined implementations of a set of image processing kernels [31]. The IF achieved an $F_{max}$ of ≈125 MHz, resulting in a peak throughput of just 24 Giga operations per second (GOPS). Compilation time was improved by 700× compared to vendor tools, with an additional FPGA resource cost of ≈40% ( 80K LUTs). The IF was subsequently mapped to a Xilinx XC5VLX330, along with a low overhead version of the interconnect with a channel width (CW) of two [38]. The original overlay used 91K LUTs and achieved an $F_{max}$ of 131 MHz while the low overhead version used 50K LUTs with an $F_{max}$ of 148 MHz, resulting in LUT/FU ratios of 465 and 255, respectively.

*2) DySER Architecture:* DySER [39], [40] was designed as a heterogeneous array of 64 functional units interconnected with a programmable network. The DySER RTL was integrated with the OpenSPARC T1 RTL and synthesized to ASIC, demonstrating a reduction in energy consumption of up to 70% and a speedup in application execution of up to 10×. The RTL of the DySER architecture was improved by using homogeneous programmable FUs and along with the OpenSPARC T1 RTL was implemented on a Xilinx Virtex-5 XC5VLX110T [33]. Due to excessive LUT consumption, it was only possible to fit a 2×2 32-bit DySER, a 4×4 8-bit DySER, or an 8×8 2-bit DySER, on the FPGA. An adapted version of a 6×6 16-bit DySER was implemented on a Xilinx Zynq XC7Z020 [41] by using DSP blocks as the FU. A benchmark set of 8 simple compute kernels with up to 23 operations required a 5×5 DySER, consuming 34K LUTs (64% of the available LUTs) on the Zynq, resulting in a LUT/FU count of 1360.

*3) DSP Block Overlays:* Many early SC overlays were developed with little consideration for the underlying FPGA architecture. The presence of hard DSP rich FPGA fabrics in modern devices, and previous work [30] that demonstrated how DSP blocks can be used for general processing at near to their theoretical limits, suggested that DSP blocks should be used as FUs to improve overlay resource usage. A fully pipelined DSP block based throughput oriented overlay architecture [34] was mapped to a Xilinx Zynq XC7Z020 device. The overlay uses the dynamic programmability of the DSP block and maps up to three operations to each node (1 add/sub, 1 mul, 1 ALU op), resulting in a significant reduction in the number of processing nodes required. The Zynq fabric is able to accommodate an 8×8 overlay consuming 28K LUTs (52% of the available LUTs) with an $F_{max}$ of 338 MHz, a peak throughput of 65 GOPS and a LUT/FU count of 437. A 2× improvement in peak throughput was demonstrated using 2 DSP blocks per FU [42].

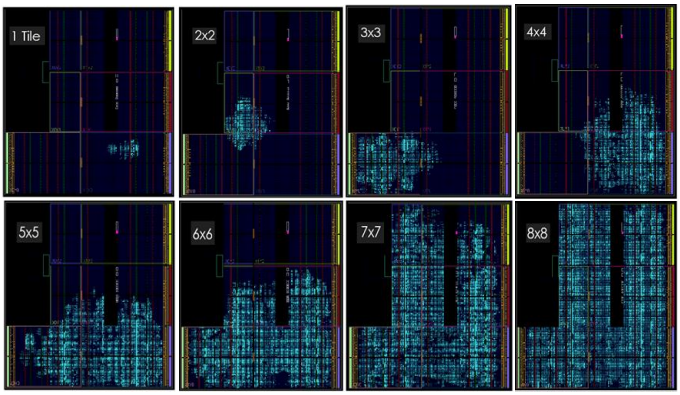


Fig. 4: Physical mapping of overlay on Zynq Fabric.

## IV. Mapping Overlays onto Zynq and Analysis

The Xilinx Zynq-7020 fabric consists of 220 DSP blocks, with a theoretical maximum frequency of 400 MHz, and each of these can support up to 3 arithmetic operations, resulting in a peak throughput of 264 GOPS. To compare the various overlays, we map the largest array possible to the Zynq device and compare the resource utilization and peak throughput. We observe that for the DySER [41] overlay, it is possible to fit an array of 36 DSP blocks (16% of the total DSP blocks), while for the 1-DSP/FU [34] and 2-DSP/FU [42] DSP block based overlays it is possible to fit 64 (30%) and 128 (60%) DSP blocks, respectively. In terms of the Peak GOPS, DySER achieves 6.3 GOPS, while the 1-DSP/FU and 2-DSP/FU overlays achieve 65 GOPS and 115 GOPS, representing 2.4%, 25% and 44% of the maximum achievable GOPS, respectively. Fig. 4 shows the mapping of the 2-DSP/FU overlay, for different array sizes (from a single tile up to an 8×8 array of tiles) onto the Zynq Fabric.

TABLE I: Quantitative Comparison of Overlays

| Resource | IF [38] | IF (opt) [38] | [41] | [34] | [42] | [42] |
|---|---|---|---|---|---|---|
| Device | XC5VLX330 | XC5VLX330 | XC7Z020 | XC7Z020 | XC7Z020 | XC7VX690T |
| Slices\|LUTs | 51.8K\|207K | 51.8K\|207K | 13.3K\|53K | 13.3K\|53K | 13.3K\|53K | 108.3K\|433.2K |
| Overlay | 14×14 | 14×14 | 6×6 | 8×8 | 8×8 | 20×20 |
| LUTs used | 91K(44%) | 50K(24%) | 48K(90%) | 28K(52%) | 37K(70%) | 228K(52%) |
| Fmax (MHz) | 131 | 148 | 175 | 338 | 300 | 380 |
| Max OPs | 196 | 196 | 36 | 192 | 384 | 2400 |
| Peak GOPS | 25.6 | 29 | 6.3 | 65 | 115 | 912 |
| LUTs/GOPS | 3550 | 1725 | 7620 | 430 | 320 | 250 |

A quantitative comparison of the DSP based overlays with others from the research literature is given in Table I. For the different overlays we compare the frequency and peak throughput of the overlay in GOPS. However, because of the different FPGA fabrics and the different overlay architectures, it is difficult to make meaningful comparisons between the different overlays. Hence, we introduce a new comparison metric: the interconnect resource used per unit peak throughput (LUTs/GOPS), which allows us to quantify the area overhead of the overlay interconnect architectures irrespective of the FU implementation. Fig. 5 shows the LUTs/GOPS, for the various

overlays, and clearly shows that the more efficient island-style overlay in [42] has a lower overhead that approaches the ideal interconnect area overhead (of 200 LUTs/GOPS) for the Zynq device. A resource balanced overlay on Zynq would consist all of the 220 DSP blocks for computations (resulting in 264 GOPS) and 53K LUTs for interconnect, resulting in 200 LUTs/GOPS.

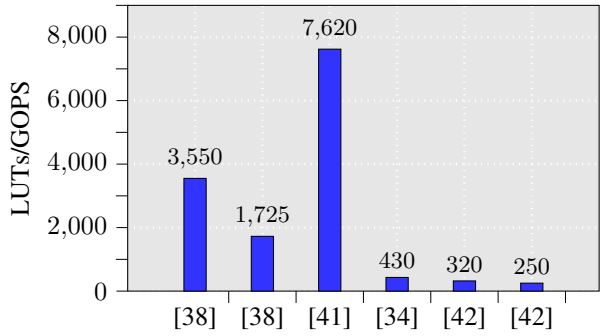


Fig. 5: Comparison of interconnect area overhead.

As a further comparison, we compare the performance, in terms of throughput per unit area, of a conventional hardware implementation with that of the overlay. To achieve this, we generate both the RTL using Vivado HLS 2013.2 and the vendor independent mapping to the overlay using our automated custom tool chain for the benchmark set described in [34]. Because the implementations we are attempting to compare use different hardware resources, it is difficult to compare them directly. Instead we normalize the hardware resource utilization using a single equivalent slices (e-Slices) metric, where we assume that 1 DSP block is equivalent to 60 slices based on the ratio of slices/DSP on the Zynq XC7Z02-1CLG484C (which is approximate 60). We observe that the average throughput per unit area for the RTL implementation of the benchmark set is $\approx$ 10 MOPS/eSlice. In comparison, the overlay in [42] achieves 2.2 MOPS/eSlice, which is around 22% of the HLS implementations. However, this 4.5$\times$ hardware performance penalty needs to be considered in context with the 1200$\times$ improvement in the place and route time and the 1000$\times$ improvement in the kernel context switch time [42], making the overlay concept a promising possibility for general purpose on-demand application acceleration.

## V. FPGA Overlays for General Purpose Application Acceleration

In the previous section, we showed that the area and performance overheads of an overlay can be reduced drastically using an architecture aware design, such that the overheads compared to a conventional hardware design are far outweighed by the design and run-time improvements. These overlays can then be combined with a host processor as a co-processor [10], [8], as in Fig. 6, with run-time management, including overlay configuration and data communication, being performed under OS [10] or hypervisor [9] control.

This provides a significant advantage over conventional FPGA accelerators as it now allows the use of multiple independent accelerators, which can be very quickly mapped to the overlay on demand, with software-like context switch times, as the application runs. Due to long FPGA configuration times, conventional FPGA accelerators usually require all accelerator cores to be present on the FPGA fabric. This results in the need for a large FPGA device, negating any power and cost advantages associated with the use of these hardware accelerators. Even if using dynamic partial reconfiguration, the delay in swapping between accelerator implementations, is much too slow for many applications.

As an example, consider the modified CODEZERO hypervisor in [8], running on the dual core ARM processor of the Xilinx Zynq, which was able to support multiple hardware and software tasks running in different hypervisor containers. Using this hypervisor with the 5$\times$5 overlay in [34], compute intensive parts of the application can be offloaded in a transparent manner using the system shown in Fig. 6. For this system, the programming and execution model is as follows: First, the user profiles the application to identify code hot-spots that can benefit from hardware acceleration (that is, identify the kernels). Then they perform hardware software partitioning, modifying the code, by inserting overlay API calls, so that those portions run on the overlay rather than the processor. Then, a compiler front-end, such as LLVM Clang, is used to convert the kernel into a machine independent optimized intermediate representation. Next, an FU-aware data flow graph (DFG) is generated, after code restructuring and additional loop-specific optimizations. Finally, a place and route tool is used to map the DFG to the overlay, producing a configuration that is used to program the overlay to implement the kernel. In this way, the overlay acts as a streaming accelerator, with data streamed from input BRAMs through the overlay back to output BRAMs.

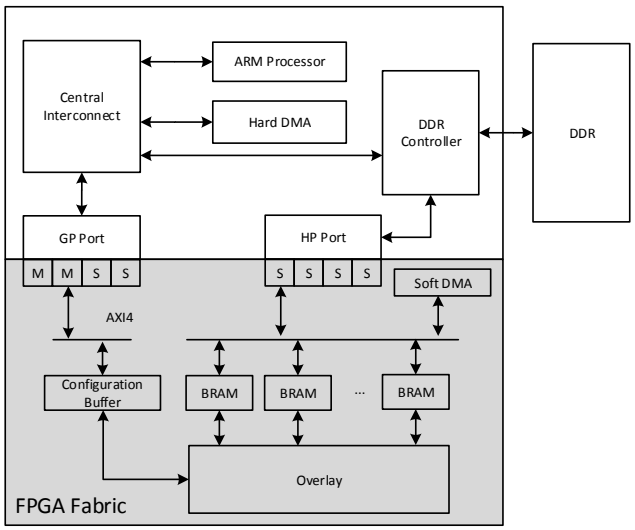


Fig. 6: Block Diagram of the System.

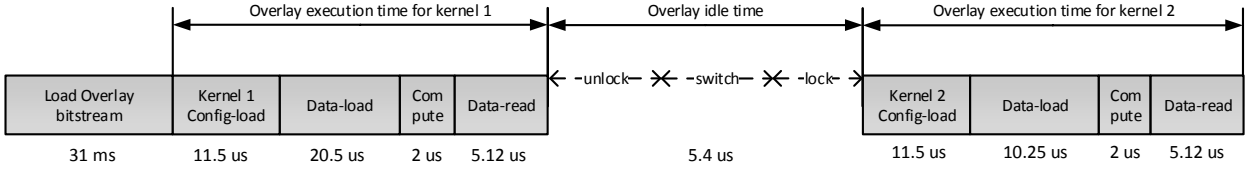


Fig. 7: Execution profile of tasks on overlay.

To demonstrate this process, we use an example scenario where two application kernels (*FFT* and *Kmeans*) are required to execute in a sequential manner, as shown in Fig. 7. Initially, the FPGA bitstream describing the overlay, the BRAM memory, the configuration buffer and an FPGA-based Xilinx soft-core DMA engine are loaded (once only) at power-on as the hypervisor is booting. To support both kernel configurations, we use four dual-port input BRAMs and a single dual-port output BRAM, configured as a 512x64-bit memory. The DMA engine uses a single 64-bit HP port on the ARM-based processor system (PS). The total configuration time is approximately 31 ms on the Zynq.

The hypervisor then schedules the first kernel (*Kmeans*) to the overlay. The configuration size for the overlay is 287 Bytes (independent of the kernel) which when sent to the configuration buffer via the GP port takes 11.5 us. The *Kmeans* kernel has sixteen 16-bit inputs and one 16-bit output, requiring all four 64-bit wide input BRAMs and one quarter of the 64-bit output BRAM. Thus for each computation, we transfer 16KB of input data and 1KB of output data. Using the Xilinx soft-core DMA engine, it takes 5.12 us to transfer the data from one BRAM to the external memory. Hence input data transfer takes 20.5 us while output data transfer takes 5.12 us (as for simplicity, we transfer the full BRAM contents). For the $5\times5$ overlay operating at 250 MHz, it takes approximately 2 us to process the streamed data. The data transfer process then repeats, until the task is finished or the *Kmeans* kernel is preempted.

Upon kernel preemption, the hypervisor unlocks the overlay, performs a hardware context switch and locks it for the next task, which takes 5.4 us (the worst case when the containers having *FFT* and *Kmeans* are both running on the same core). The hypervisor then schedules the second kernel (*FFT*) to the overlay, which again requires 287 Bytes to be sent to the configuration buffer and takes 11.5 us. The *FFT* kernel has six 16-bit inputs and four 16-bit outputs, requiring that we allocate two of the four 64-bit wide input BRAMs and the full 64-bit wide output BRAM, again allowing us to process 512 data packets. Hence input data transfer takes 10.25 us, data processing takes 2 us, while output data transfer takes 5.12 us. Again, the data transfer process repeats until the task is finished or the *FFT* kernel is preempted.

In summary, as shown in Fig. 7, it takes approximately 31 ms to configure the FPGA with the overlay infrastructure at start-up. It then takes the hypervisor approximately 11.5 us to implement the *Kmeans* kernel on the overlay. The data transfer/process/transfer cycle requires approximately 27.6 us for 512 data packets. A non-preemptive hardware context switch requires approximately 5.4 us. The hypervisor then implements the *FFT* kernel which takes approximately 11.5 us, followed by the *FFT* data transfer/process/transfer cycle which requires approximately 17.4 us for 512 data packets.

From this example, it can be seen that the time to configure the overlay, perform a hardware context switch, and reconfigure the overlay for the next kernel is relatively insignificant (if processing more than a single 512-deep data packet). However, a closer examination of the data transfer/process/transfer cycle shows that the DMA based data transfer is a major bottleneck, with data transfer representing $\approx$ 93% of the cycle time for the *Kmeans* kernel. This is using a relatively fast FPGA soft core DMA engine which is 4–5$\times$ faster than the hard DMA associated with the ARM-based processor system. This transfer time could be improved by replicating DMA controllers and using all four HP ports, overlapping computation with communication or by implementing a streaming interface directly to/from the FPGA using PCIe interfaces. However, the important point to take away from this experiment is the overlay is not the bottleneck that it once was, and is now able to adequately support general purpose hardware acceleration on FPGA.

## VI. Conclusion

We have examined the use of overlays, a virtual abstraction on top of the conventional FPGA fabric, for general purpose on-demand application acceleration. We first presented an efficient spatially configured overlay, with FUs implemented using DSP blocks, which is better able to target the underlying FPGA architecture. We showed that this overlay had a 4.5$\times$ hardware performance penalty compared to a conventional hardware implementation, but was able to achieve a 1200$\times$ improvement in the place and route time and the 1000$\times$ improvement in the hardware kernel context switch time, making it a promising possibility for general purpose on-demand application acceleration. We then explored embedding the DSP-based overlay within a heterogeneous FPGA platform, along with a modified hypervisor, for use as a rapidly reconfigurable general purpose accelerator. Here we saw that even with an efficient DMA controller, data transfer, and not the overlay, was the bottleneck, clearly showing the benefits of an overlay for supporting hardware acceleration of tasks. In the future, we plan to investigate techniques for overcoming the data communication bottleneck, and examine the power and cost benefits of accelerator overlays.

## REFERENCES


[1] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[2] S. Ahmad, V. Boppana, I. Ganusov, V. Kathail, V. Rajagopalan, and R. Wittig, "A 16-nm multiprocessing system-on-chip field-programmable gate array platform," *IEEE Micro*, vol. 36, no. 2, pp. 48–62, 2016.

[3] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable computing architectures," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 332–354, 2015.

[4] S. M. Trimberger, "Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 318–331, 2015.

[5] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, April 2011.

[6] G. Stitt, "Are field-programmable gate arrays ready for the mainstream?" *IEEE Micro*, vol. 31(6), pp. 58–63, 2011.

[7] N. W. Bergmann, S. K. Shukla, and J. Becker, "QUKU: a dual-layer reconfigurable architecture," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 1s, pp. 63:1–63:26, Mar. 2013.

[8] 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.*, vol. 77, no. 1–2, pp. 61–76, 2014.

[9] K. D. Pham, A. K. Jain, J. Cui, S. A. Fahmy, and D. L. Maskell, "Microkernel hypervisor for a hybrid ARM-FPGA platform," in *Proceedings of the International Conference on Application-Specific Systems, Architecture Processors (ASAP)*, 2013, pp. 219–226.

[10] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou, "A fully pipelined and dynamically composable architecture of CGRA," in *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 2014.

[11] A. DeHon, "DPGA utilization and application," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 1996, pp. 115–121.

[12] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A time-multiplexed FPGA," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 1997, pp. 22–28.

[13] K. Vipin and S. A. Fahmy, "Mapping adaptive hardware systems with partial reconfiguration using CoPR for Zynq," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, June 2015, pp. 1–8.

[14] K. Vipin and S. A. Fahmy, "Architecture-aware reconfiguration-centric floorplanning for partial reconfiguration," in *Proceedings of the International Symposium on Applied Reconfigurable Computing (ARC)*, 2012, pp. 13–25.

[15] K. Vipin and S. A. Fahmy, "A high speed open source controller for FPGA partial reconfiguration," in *Proceedings of International Conference on Field Programmable Technology (FPT)*, 2012, pp. 61–66.

[16] K. Vipin and S. A. Fahmy, "ZyCAP: Efficient partial reconfiguration management on the Xilinx Zynq," *IEEE Embedded Systems Letters*, Jan. 2014.

[17] T. Callahan, J. Hauser, and J. Wawrzynek, "The Garp architecture and C compiler," *Computer*, vol. 33, no. 4, pp. 62–69, Apr. 2000.

[18] E. Mirsky and A. DeHon, "MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Apr. 1996, pp. 157–166.

[19] C. Ebeling, D. C. Cronquist, and P. Franklin, "RaPiD - reconfigurable pipelined datapath," in *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, 1996, pp. 126–135.

[20] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, 2000.

[21] J. M. P. Cardoso and M. Weinhardt, "XPP-VC: a c compiler with temporal partitioning for the PACT-XPP architecture," in *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, Jan. 2002, pp. 864–874.

[22] P. Heysters and G. Smit, "Mapping of DSP algorithms on the MONTIUM architecture," in *Parallel and Distributed Processing Symposium*, 2003.

[23] C. Liang and X. Huang, "SmartCell: an energy efficient coarse-grained reconfigurable architecture for stream-based applications," *EURASIP Journal on Embedded Systems*, vol. 2009, no. 1, pp. 518–659, Jun. 2009.

[24] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "ADRES: an architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *Field Programmable Logic and Application*, Jan. 2003, pp. 61–70.

[25] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, "SPR: an architecture-adaptive CGRA mapping tool," in *Proceedings of the International Symposium on Field programmable gate arrays (FPGA)*, 2009, pp. 191–200.

[26] T. Miyamori and K. Olukotun, "REMARC: reconfigurable multimedia array coprocessor," in *IEICE Transactions on Information and Systems*, vol. 82, no. 2, 1999, pp. 389–397.

[27] N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. Wilson, M. Wrighton, and A. DeHon, "Packet switched vs. time multiplexed FPGA overlay networks," in *IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, 2006.

[28] C. Liu, H.-C. Ng, and H. K.-H. So, "QuickDough: a rapid fpga loop accelerator design framework using soft CGRA overlay," in *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, 2015.

[29] H.-P. Rosinger, "Connecting customized ip to the microblaze soft processor using the fast simplex link (fsl) channel," *Xilinx Application Note*, 2004.

[30] 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)*, vol. 7, no. 3, pp. 19:1–19:23, 2014.

[31] G. Stitt and J. Coole, "Intermediate fabrics: Virtual architectures for near-instant FPGA compilation," *IEEE ESL*, vol. 3(3), pp. 81–84, 2011.

[32] D. Capalija and T. S. Abdelrahman, "A high-performance overlay architecture for pipelined execution of data flow graphs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2013.

[33] J. Benson, R. Cofell, C. Frericks, C.-H. Ho, V. Govindaraju, T. Nowatzki, and K. Sankaralingam, "Design, integration and implementation of the DySER hardware accelerator into OpenSPARC," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2012.

[34] A. K. Jain, S. A. Fahmy, and D. L. Maskell, "Efficient Overlay architecture based on DSP blocks," in *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 2015, pp. 25–28.

[35] J. Coole and G. Stitt, "Adjustable-cost overlays for runtime compilation," in *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 2015, pp. 21–24.

[36] D. Capalija and T. Abdelrahman, "Towards synthesis-free JIT compilation to commodity FPGAs," in *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 2011.

[37] J. Coole and G. Stitt, "Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing," in *Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, 2010, pp. 13–22.

[38] A. Landy and G. Stitt, "A low-overhead interconnect architecture for virtual reconfigurable fabrics," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2012, pp. 111–120.

[39] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2011, pp. 503–514.

[40] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, 2012.

[41] A. K. Jain, X. Li, S. A. Fahmy, and D. L. Maskell, "Adapting the DySER architecture with DSP blocks as an overlay for the Xilinx Zynq," *SIGARCH Comput. Archit. News*, vol. 43, no. 4, pp. 28–33, Apr. 2016.

[42] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Throughput oriented FPGA overlays using DSP blocks," in *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, 2016, pp. 1628–1633.