Microkernel Hypervisor for a Hybrid ARM-FPGA Platform

Khoa Dang Pham*, Abhishek Kumar Jain*, Jin Cui†, Suhaib A. Fahmy*, Douglas L. Maskell*

*School of Computer Engineering, Nanyang Technological University, Singapore
{khoa002, abhishek013}@e.ntu.edu.sg, {sfahmy, asdouglas}@ntu.edu.sg

†TUM CREATE Centre for Electromobility, Singapore
jin.cui@tum-create.edu.sg

***Abstract*—Reconfigurable architectures have found use in a wide range of application domains, but mostly as static accelerators for computationally intensive functions. Commodity computing adoption has not taken off due primarily to design complexity challenges. Yet reconfigurable architectures offer significant advantages in terms of sharing hardware between distinct isolated tasks, under tight time constraints. Trends towards amalgamation of computing resources in the automotive and aviation domains have so far been limited to non-critical systems, because processor approaches suffer from a lack of predictability and isolation. Hybrid reconfigurable platforms may provide a promising solution to this, by allowing physically isolated access to hardware resources, and support for computationally demanding applications, but with improved programmability and management. We propose virtualized execution and management of software and hardware tasks using a microkernel-based hypervisor running on a commercial hybrid computing platform (the Xilinx Zynq). We demonstrate a framework based on the CODEZERO hypervisor, which has been modified to leverage the capabilities of the FPGA fabric. It supports discrete hardware accelerators, dynamically reconfigurable regions, and regions of virtual fabric, allowing for application isolation and simpler use of hardware resources. A case study demonstrating multiple independent (and isolated) software and hardware tasks is presented.**

Keywords*—*Reconfigurable systems, hypervisor, virtualization, field programmable gate arrays.

I. Introduction

Reconfigurable computing has the potential to provide a computing performance and power advantage compared to processor based systems [1]. However, while reconfigurable platforms have established themselves in specific application areas, such as the DSP and communications domains, widespread usage has been limited by poor design productivity [2]. To unleash the power of reconfigurable computing and allow spatially programmable architectures to play a full-featured role alongside general purpose processors, it is necessary to exploit the key advantages of reconfigurable hardware while abstracting implementation details to facilitate scaling. In order to solve these problems, we propose a computing model that abstracts hardware details such as spatial placement, hardware structure and device capacity using existing virtualization techniques.

Virtualization of computing components (e.g. hardware platforms, operating systems (OS), storage and network devices, etc.) in conventional processor based computing systems is well established, particularly in workstation and server environments. This is because virtualization enables a diversity of service capabilities across different OSs on a unified physical platform. One of the best examples of virtualization is cloud computing. These concepts are also being extended to embedded systems equipped with increasingly complex low-power microprocessors. For example, embedded virtualization is already used in smartphones and is being used to consolidate multiple processors and controllers in vehicles [3].

Traditional virtualization techniques for both mainstream computing and embedded systems generally only consider conventional computing resources, and do not apply to hybrid computing resources. While hardware-assisted virtualization is commonplace [4], virtualization of hardware-based computing, such as FPGA fabric, is not. Their reconfiguration capability means FPGAs can be considered as a shared compute resource (similar to a CPU) allowing multiple hardware-based tasks to complete in a time-multiplexed manner. At this level, the FPGA should not just be considered a coprocessor designed for static behaviour, but rather, it should adapt to changing processing requirements. FPGA virtualization can also improve designer productivity by abstracting FPGA resources and reducing the gap between high level synthesis tools and fine-grained FPGA architecture [5]. Virtualizing a HW-SW hybrid computing system can compensate for some of the drawbacks of hardware platforms while improving overall performance.

In this paper, we examine FPGA virtualization on a hybrid computing platform (the Xilinx Zynq 7000), by integrating virtualization of the FPGA fabric into a traditional hypervisor (CODEZERO from B Labs). We present a prototype virtualization architecture with support for HW-SW task management with the following features:

- A virtualised hybrid architecture based on an intermediate fabric (IF) built on top of the Xilinx Zynq platform.
- A hypervisor, based on the CODEZERO hypervisor, which provides secure hardware and software containers ensuring full hardware isolation between tasks.
- Efficient HW-SW communication mechanisms integrated into the hypervisor API.
- A hypervisor based context switch and scheduling mechanism for both hardware and software tasks.

The remainder of the paper is organized as follows: Section II examines current state of the art in virtualization of reconfigurable systems. Section III introduces the hypervisor

framework, development flow, and its components. In section IV we present and describe the virtualized execution and scheduling of hardware tasks on the proposed platform, HW-SW communication and the context-switch mechanisms. In Section V, we present a case study which demonstrates the basic capabilities of this approach. We conclude in Section VI and discuss some of our future work.

II. Related Work

The concept of hardware virtualization has existed since the early 1990s, when several reconfigurable architectures were proposed in [6], [7]. These architectures allowed for isolation (often referred to as virtualization) in the execution of tasks on a reconfigurable fabric. Currently, there is significant ongoing research in the area of hardware virtualization. To facilitate the virtualized execution of SW and HW tasks on reconfigurable platforms, a number of important research questions relating to the hardware aspects of virtualization must be addressed. These include:

- Rapid high-level synthesis and implementation of applications into hardware
- Rapid partial reconfiguration of the hardware fabric to support application multiplexing
- Maximising data transfer between memory/processor and the reconfigurable fabric
- Efficient OS/hypervisor support to provide task isolation, scheduling, replacement strategies, etc.

Initial implementations of dynamic reconfiguration [6], [7] required the reconfiguration of the complete hardware fabric. This resulted in significant configuration overhead, which severely limited their usefulness. Xilinx introduced the concept of dynamic partial reconfiguration (DPR) which reduced the configuration time by allowing a smaller region of the fabric to be dynamically reconfigured at runtime. DPR significantly improved reconfiguration performance [8], however the efficiency of the traditional design approach for DPR is heavily impacted by how a design is partitioned and floorplanned [9], [10], tasks that require FPGA expertise. Furthermore, the commonly used configuration mechanism is highly sub-optimal in terms of throughput [11]. In a virtualized environment, DPR would be performed under the control of the hypervisor (or OS), and would require maximum configuration throughput using the Internal Configuration Access Port (ICAP).

High-level synthesis [12] has been proposed as a technique for addressing the limited design productivity and manpower capabilities associated with hardware design. However, the long compilation times associated with synthesis and hardware mapping (including place and route) have somewhat limited these techniques to static reconfigurable systems. To address this shortcoming, significant research effort has been expended in improving the translation and mapping of applications to hardware. Warp [13] focused on fast place and route algorithms, and was used to dynamically transform executing binary kernels into customized FPGA circuits, resulting in significant speedup compared to the same kernels executing on a microprocessor. To better support rapid compilation to hardware, coarse grained architectures [14] and overlay networks [15] have been proposed. Other work has sought to maximise the use of FPGAs' heterogenous resources, such as iDEA, a procesor built on FPGA DSP blocks [16]. More recently, virtual intermediate fabrics (IFs) [5], [17] have been proposed to support rapid compilation to physical hardware. Alternatively, the use of hard macros [18] has been proposed.

Another major concern, in both static and dynamic reconfigurable systems, is data transfer bandwidth. To address possible bottleneck problems, particularly in providing high bandwidth transfers between the CPU and the reconfigurable fabric, it has been proposed to more tightly integrate the processor and the reconfigurable fabric. A number of tightly coupled architectures have resulted [19], [20], including vendor specific systems with integrated hard processors. A data-transport mechanism using a shared and scalable memory architecture for FPGA based computing devices was proposed in [21]. It assumes that the FPGA is connected directly to L2 cache or memory interconnect via memory interfaces at the boundaries of the reconfigurable fabric.

Hypervisor or OS support is crucial to supporting hardware virtualisation. A number of researchers have focused on providing OS support for reconfigurable hardware so as to provide a simple programming model to the user and effective run-time scheduling of hardware and software tasks [22], [23], [24], [25]. A technique to virtualize reconfigurable co-processors in high performance reconfigurable computing (HPRC) systems was presented in [26]. ReconOS [27] is based on an existing embedded OS (eCos) and provides an execution environment by extending a multi-threaded programming model from software to reconfigurable hardware. Several Linux extensions have also been proposed to support reconfigurable hardware [28], [29]. RAMPSoCVM [30] provides runtime support and hardware virtualization for an SoC through APIs added to Embedded Linux to provide a standard message passing interface.

To enable virtualized execution of tasks, a hybrid processor consisting of an embedded CPU and a coarse grained reconfigurable array with support for hardware virtualization, called Zippy [31] was proposed. TARTAN [32] also uses a rapidly reconfigurable, coarse-grained architecture which allows virtualization based on three aspects: runtime placement, prefetching and location resolution methods for inter-block communication. The SCORE programming model [33] was proposed to solve problems such as software survival, scalability and virtualized execution using fine-grained processing elements. However, SCORE faced major challenges due to long reconfiguration and compilation times in practice. Various preemption schemes for reconfigurable devices were compared in [34], while mechanisms for context-saving and restoring were discussed in [35] and [36].

While there has been a significant amount of work in providing OS support for hardware virtualization, this approach is less likely to be appropriate for future high performance embedded systems. For example, a modern vehicle requires a significant amount of computation, ranging from safety critical systems, through non-critical control in the passenger compartment, to entertainment applications. The current trend is towards amalgamation of computing resources to reduce cost pressures [37]. While it is unlikely that individual safety critical systems, such as ABS braking, would be integrated into a single powerful multicore processor, future safety critical systems with hard real-time deadlines, such as drive-by-wire or autonomous driving systems, are possible candidates for

amalgamation, possibly also with hardware acceleration. This combination of hard real-time, soft real-time and non real-time applications all competing for compute capacity on a hybrid multicore/reconfigurable platform cannot be supported by a conventional OS. In this situation, the microkernel based hypervisor is likely to be a much better candidate for embedded hardware virtualization because of the small size of the trust computing base, its software reliability, data security, flexibility, fault isolation and real-time capabilities [38], [39]. The importance of a microkernel is that it provides a minimal set of primitives to implement an OS. For example, the L4 microkernel [40] provides three key primitives to implement policies: address space, threads and inter process communication. Some examples of an L4 microkernel include PikeOS [41], OKL4 [39] and CODEZERO [42]. However, most of the existing microkernel based hypervisors focus only on virtualization of conventional computing systems and do not consider reconfigurable hardware.

III. Platform Framework

While the integration of relatively powerful multi-core processors with FPGA fabric has opened up many opportunities in the embedded systems domain, designer productivity is likely to remain an issue into the near future. To start to address this problem, we have developed a general framework for a microkernel based hypervisor to virtualize the Xilinx Zynq 7000 hybrid computing platform so as to provide an abstraction layer to the user. The CODEZERO hypervisor [42] is modified to virtualize both the hardware and software components of this platform enabling the use of the CPU for software tasks and the FPGA for hardware tasks in a relatively easy and efficient way. A block diagram of the hybrid computing platform is shown in Fig. 1.

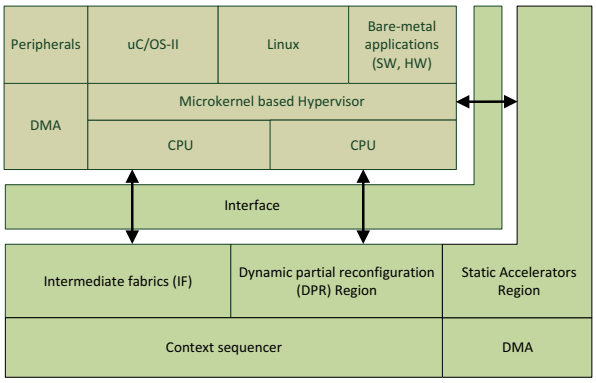

Fig. 1: Block diagram of the hybrid computing platform.

In this framework, we are able to execute a number of operating systems (including uCOS-II, Linux and Android) as well as bare metal/real-time software, each in their own isolated container. By modifying the hypervisor API (described in the next section), support for hardware acceleration can also be added, either as dedicated real-time bare metal hardware tasks, real-time HW/SW bare metal applications or HW/SW applications running under OS control. This allows the time-multiplexed execution of software and hardware tasks concurrently. In these scenarios, a hardware task corresponds to FPGA resources configured to perform a particular acceleration operation, while a software task corresponds to a traditional task running on the CPU. The framework treats the FPGA region as either a static reconfigurable region, a DPR region or a region of IF similar to those in [5], [14] or any combination of these. The hypervisor is able to dynamically modify the behaviour of the DPR and IF regions and carry out hardware and software task management, task-scheduling and context-switching.

As an example, a hardware task such as JPEG compression can be decomposed into several contexts where each context can determine the behaviour of the IF or DPR. These contexts can then be used to perform a time multiplexed execution of the task by loading context frames consecutively. These context frames can be defined as either hypervisor controlled commands for the IF region or pre-stored bitstreams for the DPR region. The context sequencer, shown in Fig. 4, is used to load context frames into these regions and also to control and monitor the execution of these hardware tasks. The context sequencer is described in more detail in the next section.

A. The Reconfigurable Fabric

A number of additional structures are needed to support hypervisor control of regions of the reconfigurable fabric, as shown in Fig. 2.

1) Task communication: The Zynq-7000 provides several AXI based interfaces to the reconfigurable fabric. Each interface consists of multiple AXI channels and hence provides a large bandwidth between memory, processor and programmable logic. The AXI interfaces to the fabric include:

- AXI_ACP – one cache coherent master port
- AXI_HP – four high performance, high bandwidth master ports
- AXI_GP – four general purpose ports (two master and two slave ports)

All of these interfaces support DMA data transfer between the fabric and main memory (at different bandwidths) as shown in Fig. 2. These different communication mechanisms can be applied for different performance requirements. For example, for a DPR region, a DMA transfer can be used to download and read-back the bitstream via the processor configuration access port (PCAP), while for an IF region, the contexts are transferred between main memory and the context frame buffer (CFB) under DMA control.

2) Context Frame Buffer: A CFB, as shown in Fig. 2, is needed to store the context frames. A HW task can be decomposed into several consecutive contexts. While the context frames and other user data for small applications could be stored in Block RAMs (BRAMs) in the fabric, this would scale poorly as the number of contexts and size of the IF increases. Hence, the CFB is implemented as a two level memory hierarchy. The main (external) memory is used to store context frames which are transferred to the CFBs (implemented as BRAMs in the FPGA) when needed, similar to the cache hierarchy in a processor.

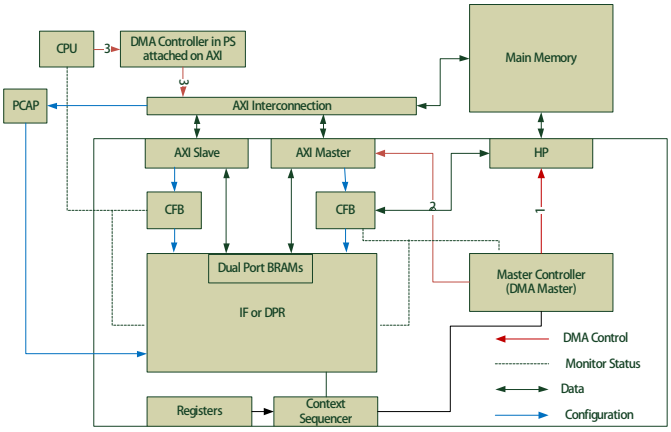

Fig. 2: Block Diagram of the Reconfigurable Region.

3) Intermediate Fabric: Use of an IF provides a much quicker mechanism for hardware task management at runtime without going through the full compilation (synthesis, map, place and route) process. This is because the behaviour of the IF can be directly modified using hypervisor controlled commands. However, mapping circuits to IFs is less efficient than using DPR or static implementations.

The IF (shown in Fig. 3) consists of programmable processing elements (PEs), programmable interconnections and BRAMs, whose behaviour is defined by the contents of a context frame. Each PE is connected to all of its 4 immediate neighbours using programmable crossbar (CB) switches. We have used a multiplexer based implementation for the CB, which can provide the data-flow direction among PEs and a cross connection between inputs and outputs. The operation of the PEs and CBs is set by PE and CB commands, respectively, in the context frame.

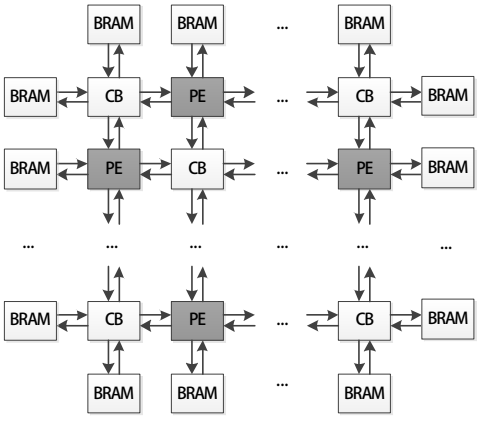

Fig. 3: Block Diagram of the Intermediate Fabric.

Dual Port BRAMs are used for data-streaming to and from the IF at a very high speed, with one port connected to the IF and the other port connected to the interconnect (e.g. an AXI general slave port). A context frame can configure all PEs, CBs, and BRAMs, and determines the data-flow, interconnection, and the number of working PEs in a context. As a result, different contexts will have different I/O latencies, which must be communicated back to the hypervisor. The context frame also includes the working mode selection for a context. We have currently implemented three modes: 1D-systolic (streaming), 2D-systolic, and dataflow. The working mode determines how the fabric operates and how the data flows into and out of the fabric. A context frame has the following command groups:

- PE commands: define the operation of a processing element.
- CB commands: define the operation of a crossbar switch.
- USER commands: set the I/O base addresses, working mode, latency, etc.
- BRAM commands: define the operation and data flow of the BRAMs.

Device drivers have been added to CODEZERO to support the IF, as shown in Table I.

TABLE I: Intermediate Fabric driver functions.

Driver function	Functionality
`gen_CF` (context_id, addr, latency, num_pe, num_cb, context_mode)	Generate a CF for a context with id to locn in mem, set latency, num of PEs num of CBs and mode (1D/2D/dataflow)
`set_CB_command` (pos, dir, mode)	Configure the direction and cross-connection for a CB in the IF
`set_PE_command` (pos, dir, op)	Configure the direction and operation of a PE in the IF
`set_BRAM_command` (pos,input/output, mode)	Configure the I/O and the data pattern mode of a BRAM in the IF
`start_IF` (addr, num_context)	Start a HW task in the IF, load the CF from the base address, and set the context number
`reset_IF` ()	Reset the IF
`set_Input_addr` (addr)	Set the start address for data/BRAM input
`set_Output_addr` (addr)	Set the start address for data/BRAM output

4) DPR Region: The DPR region provides a mechanism for hardware task management at the cost of a significant reconfiguration time overhead. This is because the DPR region can only be efficiently modified using pre-stored bit-streams (generated using vendor tools). However, DPR allows for highly customised IP cores for better performance.

IV. The Hybrid Platform Hypervisor

A microkernel-based hypervisor is a minimal OS that runs directly on the bare (CPU) hardware. The hypervisor creates an abstraction of the underlying hardware platform so that it can be used by one or more guest OSs. In this context, a guest OS and its applications (or a bare-metal application) runs within a hypervisor container which provides the necessary isolation.

A. Porting CODEZERO to the Xilinx Zynq-7000

In this section, we describe the necessary modifications to the CODEZERO hypervisor [42], firstly for it to execute on the dual-core ARM Cortex-A9 processor of the Zynq-7000 hybrid platform, and secondly, to provide hypervisor support for HW task execution and scheduling, by adding

additional mechanisms and APIs for FPGA virtualization. Currently, CODEZERO only runs on a limited number of ARM-based processors, and we first ported it to the Xilinx Zynq-7000 hybrid computing platform. The main changes needed included:

- Rewriting the drivers (e.g. PCAP, timers, interrupt controller, UART, etc.) for the Zynq-7000 specific ARM Cortex-A9 implementation
- FPGA initialization (e.g. FPGA clock frequency, I/O pin mapping, FPGA interrupt initialization, etc.)
- HW task management and scheduling
- DMA transfer support

As the first two changes are mainly technical and not specific to the requirements needed to support hardware virtualization, we will not discuss these further.

B. Context sequencer behaviour

As mentioned in Section III, a context sequencer (CS) is needed to load context frames (parts of a hardware task) into the configurable regions and to control and monitor their execution, including context switching and data-flow. We provide a memory mapped register interface (implemented in the FPGA fabric and accessible to the hypervisor via the AXI bus) for this purpose. The control register is used by the hypervisor to instruct the CS to start a HW task in either the IF or DPR regions. The control register also sets the number of contexts and the context frame base address for a HW task. The status register is used to indicate the HW task status, such as the completion of a context or of the whole HW task.

In the *IDLE* state, the CS waits for the control register's start bit to be asserted before moving to the *CONTEXT_START* state. In this state, it generates an interrupt *interrupt_start_context*, and then activates a context counter before moving to the *CONFIGURE* state. In this state, the CS loads the corresponding context frame to the CFB of the IF or to the DPR region via PCAP to configure the context's behaviour. Once finished, the CS moves to the *EXECUTE* state and starts execution of the context. In this state the CS behaves like a dataflow controller, controlling the input and output data flow depending on the working mode of the context frame. Once execution finishes, the CS moves to the *CONTEXT_FINISH* stage and generates an *interrupt_finish_context* interrupt. The CS then moves to the *RESET* state which releases the hardware fabric and sets the status register completion bit for the context. When the context counter is less than the desired number of contexts, the CS starts the next context and repeats. When the desired number of contexts is achieved, the whole HW task finishes and the CS moves to the *DONE* state. This behaviour is shown in Fig. 4.

C. Task communication

We have adopted two modes to transfer context frames and user data using DMA between the IF (or DPR region) and main memory. The first, called active DMA, uses a dedicated DMA master controller, independent of the CPU, and automatically loads data when the CFB is not full. The second, called passive DMA, uses the existing DMA controller on the AXI interconnection controlled and monitored by CPU. Passive

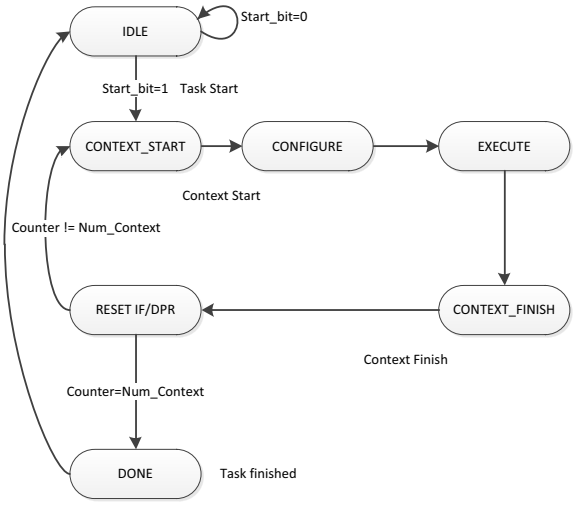

Fig. 4: State-machine based Context Sequencer.

DMA has lower throughput than active DMA. The APIs of Table II were added to support the DMA transfer modes, as well as a non-DMA transfer mode.

TABLE II: Hypervisor APIs to support DMA transfer.

APIs	Functions
`init_Active_data` (s_addr, d_addr, size)	Load user data from main memory s_addr to BRAM d_addr via DMA controller, size indicates the data block size needed to move
`start_Active_transfer` `reset_Active_transfer` `stop_Active_transfer`	These are only invoked by a fabric start, reset or stop (never by the user)
`load_CF` (addr, num_context, mode)	Load context frame from main mem addr to CFB (PCAP). mode is 1) passive DMA; 2) non-DMA (CPU): 3) active DMA
`interrupt_CFB_full`	Interrupt handler triggered when CFB is full, used for CPU monitoring the CFB status in passive DMA mode
`interrupt_PCAP_DONE`	This interrupt indicates that a bit stream downloading via DPR is done
`load_Data` (s_addr, d_addr, size, mode)	Move user data from s_addr to d_addr memory-to-BRAMs or inter-BRAM, mode 1) passive DMA, 2) non-DMA (CPU)
`poll_CFB_status`	CPU polls the CFB status and return the number of empty slots

D. Hardware task scheduling and context switching

In this section, we introduce two scheduling mechanisms to enable HW task scheduling under hypervisor control: non-preemptive hardware context switching and preemptive hardware context switching.

1) Non-preemptive hardware context switching: HW task scheduling only occurs when a HW context completes. At the start of a context (when *interrupt_start_context* is triggered), we use the hypervisor mutex mechanism (*l4_mutex_control*) to lock the reconfigurable fabric (IF or DPR) so that other

contexts cannot use the same fabric. This denotes the fabric as a critical resource in the interval of one context and can be only accessed in a mutually exclusive way. At the completion of a context (when *interrupt_finish_context* is triggered), the reconfigurable fabric lock can be released via *l4_mutex_control*. After that, a possible context switch (*l4_context_switch*) among the HW tasks can happen. The advantage of non-preemptive hardware context switching is that context saving or restoring is not necessary, as task scheduling occurs after a context finishes. Thus minimal modifications are required in the hypervisor to add support for HW task scheduling as the existing hypervisor scheduling policy and kernel scheme are satisfactory. The interrupt handlers and API modifications added to CODEZERO to support this scheduling scheme are shown in Table III.

TABLE III: Hypervisor APIs to support Hardware Task Scheduling.

APIs	Functions
`interrupt_start_context`	Triggered when every context starts. In the handler, it locks IF or DPR.
`interrupt_finish_context`	Triggered when every context finished. in the interrupt handler, it should unlock IF
`poll_Context_status` `poll_Task_status`	Poll the completion (task done) bit of a context (HW task) in the status register. Also unlocks IF (DPR) after a context finishes.

2) Pre-emptive hardware context switching: CODEZERO can be extended to support pre-emptive hardware context-switching. In this scenario, it must be possible to save a context frame and restore it. Context-saving refers to a read-back mechanism to record the current context counter (context id), the status, the DMA controller status and the internal state (e.g. the bitstream for DPR) into the thread/task control block (TCB), similar to saving the CPU register set in a context switch. The TCB is a standard data structure used by an OS or microkernel-based hypervisor. In CODEZERO this is called the user thread control block (UTCB). A context frame restore occurs when a HW task is swapped out, and an existing task resumes its operation. This approach would provide a faster response, compared to non-preemptive context switching, but the overhead (associated with saving and restoring the hardware state) is considerably higher. This requires modification of the UTCB data structure and the hypervisor's context switch (*l4_context_switch*) mechanism, as well as requiring a number of additional APIs. Pre-emptive hardware context switching is a work in progress.

V. Case Study

In this section, we present the details of a fully functioning virtualized hardware example using a simple IF operating under CODEZERO hypervisor control. In this example, the hypervisor uses three isolated containers (the term that CODEZERO uses to refer to a virtual machine), as shown in Fig. 5. The first container runs a simple RTOS (uC/OS-II) running 14 independent software tasks. The second container is a bare metal application (an application which directly accesses the hypervisor APIs and does not use a host OS) which runs an FIR filter as a hardware task. The third container is also a bare metal application which runs a hardware matrix multiplication task. The two hardware tasks are executed on the same fabric, scheduled and isolated by the hypervisor.

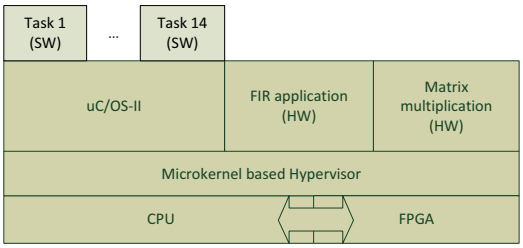

Fig. 5: Multiple Hardware and Software task scheduling.

A. Systolic FIR Filter

A simple 5-tap systolic FIR filter (shown in Fig. 6) is used for the first hardware task. This structure is composed of five processing units and it can be efficiently mapped to the IF as shown in Fig. 6 with a latency of 12 cycles. That is, the FIR application has a single context frame. The PE is configured as a DSP block with 3 inputs and 2 outputs. The FIR filter coefficients are input and stored to a PE internal register. The CBs are configured to map the data-flow as shown in Fig. 6. The input data is transferred via the AXI bus and stored in the "input" BRAM. The processed data is stored to the "output" BRAM. The output data is then read by the CPU via the AXI bus.

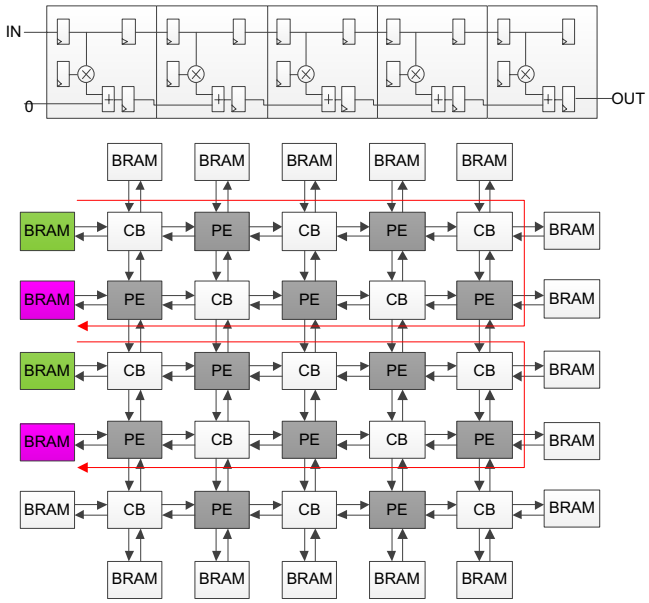

Fig. 6: Systolic FIR filter and its mapping on the IF.

B. Matrix Multiplication

The second hardware task is a matrix multiplication. Fig. 7 shows the computation of one output element C for the matrix product of matrices A and B (3×3 matrices). By mapping this

structure as a hardware task to the IF three times it is possible to calculate three output elements simultaneously, as shown in Fig. 7. Thus the complete task can finish in three such contexts. In this example, the PE is configured as a DSP block with 3 inputs and a single output. The CBs are configured to map the data-flow as shown in Fig. 7, requiring 3 "input" BRAMs and 3 "output" BRAMs. The latency of a context is 8 cycles.

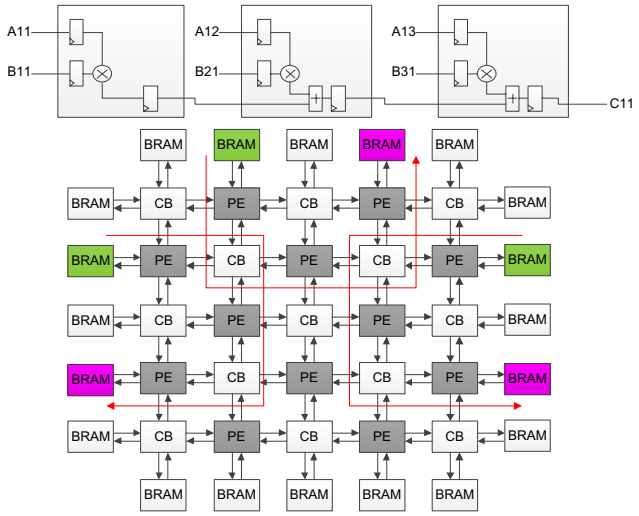

Fig. 7: Matrix Multiplication and its mapping on IF.

C. Multiple software-hardware tasks on ZynQ

In this experiment, uC/OS-II runs in container 0, while the FIR Filter and the matrix multiplication run in container 1 and 2, respectively, as shown in Fig. 5. We use the CODEZERO microkernel scheduler to switch tasks between container 0, 1 and 2. Software tasks running in container 0 are allocated and executed on the CPU. Hardware tasks running in containers 1 and 2 are allocated and run on the IF. A context of a hardware task will first lock the IF, configure the fabric behaviour, execute to completion and then unlock the fabric (that is it implements non-preemptive context switching). Algorithm 1 shows the steps involved in non-preemptive context switching. Table IV gives the hardware context switch overhead for the CODEZERO hypervisor. The context switch times are significantly less than those for Linux [43]. The configuration times and the (best-worst) hardware application response times are given in Table V. It should be noted that these times will increase both with application complexity and IF size.

TABLE IV: Hardware context switch overhead for CODEZERO.

Clock cycles (time)	Non-preemptive	Preemptive
T_{lock} (no contention)	214 (0.32μs)	NA
T_{lock} (with contention)	7738 (11.6μs)	
T_{C0_switch}	3264 (4.9μs)	3140 (4.7μs)

VI. Conclusions and Future Work

We have presented a framework for hypervisor based virtualization of both HW and SW tasks on hybrid computing architectures, such as the Xilinx Zynq 7000. The framework accommodates execution of SW tasks on the CPUs, as either real-time (or non-real-time) bare-metal applications or applications under OS control. In addition, support has been added to the hypervisor for the execution of HW tasks in the FPGA fabric, again as either bare-metal HW applications or as HW-SW partitioned applications. By facilitating the use of static hardware accelerators, partially reconfigurable modules and intermediate fabrics, a wide range of approaches to virtualization, to satisfy varied performance and programming needs, can be facilitated.

The case study demonstrates that the hypervisor functionality works, and that different types of tasks (both HW and SW) can be managed concurrently, with the hypervisor providing the necessary isolation. We are now working on providing full support for DPR, and enabling fast partial reconfiguration through the use of a custom ICAP controller and DMA bitstream transfer. Additionally, we are working on developing a more fully featured intermediate fabric, to enable higher performance and better resource use. We also plan to examine alternative communications structures between SW, memory, hypervisor and FPGA fabric, to better support virtualized HW based computing. Finally, with these initiatives we hope to reduce the hardware context switching overhead, particularly of the intermediate fabric, with the aim of developing a competitive preemptive hardware context switching approach.

TABLE V: Hardware task configuration time and total application response times for the case study.

Clock cycles (time)	Non-preemptive		Preemptive	
	FIR	MM	FIR	MM
T_{conf}	2150 (3.2μs)	3144 (4.7μs)	3392(5.1μs)	5378 (8.1μs)
T_{hw_resp}	(8.5μs-19.7μs)	(9.9μs-20.3μs)	(9.8μs)	(12.8μs)

Algorithm 1: Pseudocode for non-interrupt implementation for non-preemptive HW context switching.

```
begin
    context_id = 0;
    while (!poll_Task_status()) do
        l4_mutex_control(IF_lock, L4_MUTEX_LOCK);
        gen_CF(context_id, *(cf_base + context_id *
        sizeof(cf)), ..., );
        set_CB_commands(...);
        ...;
        set_PE_commands(...);
        ...;
        set_BRAM_commands(...);
        ...;
        set_Input_addr(*src_base);
        set_Output_addr(*dst_base);
        start_IF();
        while (!poll_Context_status()) do
        end
        reset_IF();
        context_id++;
        l4_mutex_control(IF_lock, L4_MUTEX_UNLOCK);
    end
end
```


Acknowledgement

This work was partially supported by the Singapore National Research Foundation under its Campus for Research Excellence And Technological Enterprise (CREATE) programme.

REFERENCES

[1] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Survey*, vol. 34, pp. 171–210, Jun. 2002.

[2] D. Thomas, J. Coutinho, and W. Luk, "Reconfigurable computing: Productivity and performance," in *Asilomar Conference on Signals, Systems and Computers*, 2009, pp. 685–689.

[3] S. Chakraborty, M. Lukasiewycz, C. Buckl, S. A. Fahmy, N. Chang, S. Park, Y. Kim, P. Leteinturier, and H. Adlkofer, "Embedded systems and software challenges in electric vehicles," in *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, 2012, pp. 424–429.

[4] K. Adams, "A comparison of software and hardware techniques for x86 virtualization," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[5] G. Stitt and J. Coole, "Intermediate fabrics: Virtual architectures for near-instant FPGA compilation," *IEEE Embedded Systems Letters*, vol. 3, no. 3, pp. 81–84, Sep. 2011.

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

[7] 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.

[8] M. Hubner, D. Gohringer, J. Noguera, and J. Becker, "Fast dynamic and partial reconfiguration data path with low hardware overhead on Xilinx FPGAs," in *IEEE International Symposium on Parallel Distributed Processing, Workshops and PhD Forum (IPDPSW)*, 2010, pp. 1–8.

[9] K. Vipin and S. A. Fahmy, "Automated partitioning for partial reconfiguration design of adaptive systems," in *Proceedings of IEEE International Symposium on Parallel Distributed Processing, Workshops (IPDPSW) – Reconfigurable Architectures Workshop (RAW)*, 2013.

[10] 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.

[11] 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*, 2012, pp. 61–66.

[12] Y. Liang, K. Rupnow, Y. Li, and et. al., "High-level synthesis: productivity, performance, and software constraints," *Journal of Electrical and Computer Engineering*, vol. 2012, no. 649057, pp. 1–14, Jan. 2012.

[13] F. Vahid, G. Stitt, and R. Lysecky, "Warp processing: Dynamic translation of binaries to FPGA circuits," *Computer*, vol. 41, no. 7, pp. 40–46, Jul. 2008.

[14] S. Shukla, N. W. Bergmann, and J. Becker, "QUKU: a coarse grained paradigm for FPGA," in *Proc. Dagstuhl Seminar*, 2006.

[15] 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 International Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, 2006.

[16] H. Y. Cheah, S. A. Fahmy, and D. L. Maskell, "iDEA: A DSP block based FPGA soft processor," in *Proceedings of the International Conference on Field Programmable Technology*, 2012, pp. 151–158.

[17] M. Hubner, P. Figuli, R. Girardey, D. Soudris, K. Siozios, and J. Becker, "A heterogeneous multicore system on chip with run-time reconfigurable virtual FPGA architecture," in *IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)*, 2011.

[18] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "HMFlow: accelerating FPGA compilation with hard macros for rapid prototyping," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2011.

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

[20] Z. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit," in *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2000, pp. 225–235.

[21] E. S. Chung, J. C. Hoe, and K. Mai, "CoRAM: an in-fabric memory architecture for FPGA-based computing," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays (FPGA)*, 2011, pp. 97–106.

[22] G. Brebner, "A virtual hardware operating system for the Xilinx XC6200," in *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*, 1996, pp. 327–336.

[23] C. Steiger, H. Walder, and M. Platzner, "Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1393–1407, Nov. 2004.

[24] M. Vuletic, L. Righetti, L. Pozzi, and P. Ienne, "Operating system support for interface virtualisation of reconfigurable coprocessors," in *Design, Automation and Test in Europe (DATE)*, 2004, pp. 748–749.

[25] K. Rupnow, "Operating system management of reconfigurable hardware computing systems," in *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, 2009, pp. 477–478.

[26] I. Gonzalez and S. Lopez-Buedo, "Virtualization of reconfigurable coprocessors in HPRC systems with multicore architecture," *Journal of Systems Architecture*, vol. 58, no. 6–7, pp. 247–256, Jun. 2012.

[27] E. Lübbers and M. Platzner, "ReconOS: multithreaded programming for reconfigurable computers," *ACM Transactions on Embedded Computing Systems*, vol. 9, no. 1, Oct. 2009.

[28] H. So, A. Tkachenko, and R. Brodersen, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH," in *Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2006, pp. 259–264.

[29] K. Kosciuszkiewicz, F. Morgan, and K. Kepa, "Run-time management of reconfigurable hardware tasks using embedded linux," in *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, 2007, pp. 209–215.

[30] D. Gohringer, S. Werner, M. Hubner, and J. Becker, "RAMPSoCVM: runtime support and hardware virtualization for a runtime adaptive MPSoC," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2011.

[31] C. Plessl and M. Platzner, "Zippy - a coarse-grained reconfigurable array with support for hardware virtualization," in *IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP)*, 2005, pp. 213–218.

[32] M. Mishra and S. Goldstein, "Virtualization on the tartan reconfigurable architecture," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2007, pp. 323–330.

[33] A. DeHon, Y. Markovsky, E. Caspi, M. Chu, R. Huang, S. Perissakis, L. Pozzi, J. Yeh, and J. Wawrzynek, "Stream computations organized for reconfigurable execution," *Microprocessors and Microsystems*, vol. 30, no. 6, pp. 334–354, 2006.

[34] K. Jozwik, H. Tomiyama, S. Honda, and H. Takada, "A novel mechanism for effective hardware task preemption in dynamically reconfigurable systems," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2010.

[35] H. Kalte and M. Porrmann, "Context saving and restoring for multitasking in reconfigurable systems," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2005, pp. 223–228.

[36] K. Rupnow, W. Fu, and K. Compton, "Block, drop or roll(back): Alternative preemption methods for RH multi-tasking," in *IEEE Symposium on Field Programmable Custom Computing Machines*, 2009, pp. 63–70.

[37] S. Shreejith, S. A. Fahmy, and M. Lukasiewycz, "Reconfigurable computing in next-generation automotive networks," *IEEE Embedded Systems Letters*, vol. 5, no. 1, pp. 12–15, 2013.

[38] G. Heiser, V. Uhlig, and J. LeVasseur, "Are virtual-machine monitors microkernels done right?" *ACM SIGOPS Operating Systems Review*, vol. 40, no. 1, pp. 95–99, Jan. 2006.

[39] G. Heiser and B. Leslie, "The OKL4 microvisor: convergence point of microkernels and hypervisors," in *Proceedings of the ACM Asia Pacific Workshop on Systems*, 2010, pp. 19–24.

[40] J. Liedtke, "On micro-kernel construction," in *Proceedings of the ACM Symposium on Operating Systems Principles*, 1995, pp. 237–250.

[41] R. Kaiser and S. Wagner, "Evolution of the PikeOS microkernel," in *National ICT Australia*, 2007.

[42] "Codezero project overview," in *http://dev.b-labs.com/*.

[43] F. M. David, J. C. Carlyle, and R. H. Campbell, "Context switch overheads for Linux on ARM platforms," in *Proceedings of the Workshop on Experimental Computer Science*, 2007.