J Sign Process Syst (2014) 77:61–76
DOI 10.1007/s11265-014-0884-1

Virtualized Execution and Management of Hardware Tasks on a Hybrid ARM-FPGA Platform

Abhishek Kumar Jain · Khoa Dang Pham · Jin Cui · Suhaib A. Fahmy · Douglas L. Maskell

Received: 13 September 2013 / Revised: 19 January 2014 / Accepted: 14 March 2014 / Published online: 31 May 2014

Abstract Emerging hybrid reconfigurable platforms tightly couple capable processors with high performance reconfigurable fabrics. This promises to move the focus of reconfigurable computing systems from static accelerators to a more software oriented view, where reconfiguration is a key enabler for exploiting the available resources. This requires a revised look at how to manage the execution of such hardware tasks within a processor-based system, and in doing so, how to virtualize the resources to ensure isolation and predictability. This view is further supported by trends towards amalgamation of computation in the automotive and avionics domains, where such properties are essential to overall system reliability. We present the virtualized execution and management of software and hardware tasks using a microkernel-based hypervisor running on a commercial hybrid computing platform (the Xilinx Zynq). The CODEZERO hypervisor has been modified to leverage the capabilities of the FPGA fabric, with support for discrete hardware accelerators, dynamically reconfigurable regions, and regions of virtual fabric. We characterise the communication overheads in such a hybrid system to motivate the importance of lean management, before quantifying the context switch overhead of the hypervisor approach. We then compare the resulting idle time for a standard Linux implementation and the proposed hypervisor method, showing two orders of magnitude improved performance with the hypervisor.

Keywords Reconfigurable systems · Hypervisor · Virtualization · Field programmable gate arrays

1 Introduction

While the performance benefits of reconfigurable computing over processor based systems have been well established [9], such platforms have not seen wide use beyond specialist application domains such as digital signal processing and communications. Poor design productivity has been a key limiting factor, restricting their effective use to experts in hardware design [39]. At the same time, their rapidly increasing logic density and more capable resources makes them applicable to a wider range of domains. For reconfigurable architectures to play a full-featured role alongside general purpose processors, it is essential that their key advantages be available in an abstracted manner that enables scaling, and makes them accessible to a wider audience. Our work explores how virtualization techniques can be used to help bridge this design gap, by abstracting the complexity away from the designer, enabling reconfigurable architectures to be exploited alongside general purpose processors, within a software-centric runtime framework. This new design methodology approaches the fabric from the perspective of software-managed hardware tasks, enabling more shared use, while ensuring isolation and predictability.

Virtualization is a widespread technique in conventional processor based computing systems, particularly in workstation and server environments. It enables a diversity of

A. K. Jain (✉) · K. D. Pham · S. A. Fahmy · D. L. Maskell
School of Computer Engineering,
Nanyang Technological University, Singapore, Singapore
e-mail: abhishek013@e.ntu.edu.sg

J. Cui
TUM CREATE Centre for Electromobility, Singapore, Singapore

service capabilities across different OSs on a unified physical platform, with abstraction enabling scaling and portability. 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.

Such techniques are also emerging in embedded systems where low-power multicore processors are becoming more widespread, such as in smartphones. The technology is also being explored as a way of consolidating the complex distributed computing in vehicles onto fewer nodes [6].

So far, virtualization has focused primarily on conventional computing resources. While hardware-assisted virtualization is commonplace [2], virtualization of hardware-based computing has often focused on a hardware centric view. The reconfiguration capability of FPGAs means the logic fabric can be treated as a shared compute resource (similar to a CPU), allowing multiple hardware-based tasks to make use of it in a time-multiplexed manner. Hence, the FPGA should not just be considered a static coprocessor, but rather, it should adapt to changing processing requirements.

Within the reconfigurable computing community, there have been some efforts to tackle these problems from the hardware perspective. CoRAM [8], LEAP [3], QUKU [35], intermediate fabrics [38] and VirtualRC [23] are examples of approaches to virtualizing memory and logic in reconfigurable devices. New hybrid architectures, that combine capable multicore processors with modern FPGA fabrics, represent an ideal platform for virtualized systems, since it is expected that the hardware fabrics will be used within a more software-centric environment, and a tighter coupling means regular swapping of tasks is feasible due to reduced response times.

In this paper, we present a virtualized execution platform that not only abstracts hardware details, such as reconfigurable resources (logic, memory and I/O interfaces) and their placement, but also provides system support for virtualized execution of software and hardware tasks. Our prototype system uses a Xilinx Zynq 7000 hybrid computing platform and integrates virtualization of the FPGA fabric into a traditional hypervisor (CODEZERO from B Labs). The system provides support for hardware and software 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 hardware-software communication mechanisms integrated into the hypervisor API.
- A hypervisor based context switch and scheduling mechanism for both hardware and software tasks.

We investigate the communication overhead in such a system to motivate the need for lightweight task management. We then characterise the context switching overhead of the hypervisor, before demonstrating the resulting advantage over management of hardware tasks within an operating system like Linux.

The remainder of the paper is organized as follows: Section 2 examines the current state of the art in virtualization of reconfigurable systems. Section 3 introduces the hybrid platform, its components and communication mechanisms between these components. Reconfiguration management strategies are described in Section 4 by introducing the intermediate fabric and its internal architecture. In Section 5, we present and describe the hybrid hypervisor, scheduling of hardware tasks on the proposed platform, and the context-switch mechanisms. In Section 6, we present experimental results and characterise the overheads associated with virtualized execution. A case study which demonstrates the basic capabilities of this approach has also been presented. We conclude in Section 7 and discuss some of our future work.

2 Related Work

The concept of hardware virtualization has existed since the early 1990s, when several reconfigurable architectures were proposed in [11, 40]. 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 [11, 40] 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 [18], however the efficiency of the traditional design approach for DPR is heavily impacted by how a design is partitioned and floorplanned [42, 44], tasks that require FPGA expertise. Furthermore, the commonly used configuration mechanism is highly sub-optimal in terms of throughput [43]. 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 [26] 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 [41] 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 [35] and overlay networks [22] 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 [7]. More recently, virtual intermediate fabrics (IFs) [17, 38] have been proposed to support rapid compilation to physical hardware. Alternatively, the use of hard macros [25] 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 [5, 47], 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 [8]. 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 virtualization. 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 [4, 32, 37, 45]. A technique to virtualize reconfigurable co-processors in high performance reconfigurable computing systems was presented in [14]. ReconOS [28] 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 [24, 36]. RAMPSoCVM [13] 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 [29] 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 [12] was proposed to solve problems such as software survival, scalability and virtualized execution using fine-grained processing elements. However, SCORE faced major practical challenges due to long reconfiguration and compilation times. Various preemption schemes for reconfigurable devices were compared in [19], while mechanisms for context-saving and restoring were discussed in [21] and [33].

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 [34]. 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 [15, 16]. The importance of a microkernel is that it provides a minimal set of primitives to implement an OS. For example, the L4 microkernel [27] provides three key primitives to implement policies: address space, threads and inter process communication. Some examples of an L4 microkernel include Pike-OS [20], OKL4 [15] and CODEZERO [1].

Existing microkernel based hypervisors mostly only focus on virtualization of conventional computing systems and do not consider reconfigurable hardware. An example of para-virtualization of an FPGA based accelerator as a shared resource on an x86 platform (pvFPGA) was presented in [46]. Here the Xen Hypervisor was used to share the FPGA as a static accelerator among multiple virtual machines.

In our work we use a microkernel based hypervisor on a hybrid platform and modify it to use the FPGA fabric as a dynamically reconfigurable shared resource for task execution. To the best of our knowledge this is the first microkernel based hypervisor which considers the FPGA as a shared and dynamically reconfigurable resource for task execution. The initial version of our hypervisor based implementation for HW and SW task management on a hybrid platform was presented in [30]. This paper extends this work by providing the following additional material:

- A more detailed description of the platform and its components is presented
- An overview of run time reconfiguration management using an intermediate fabric (IF), including the internal architecture of the IF components
- A case-study to analyze HW-SW communication overheads
- A comparison of the context switch overheads of Linux and the proposed hypervisor based system when running hardware tasks.

3 The Hybrid Platform (Hardware)

Both major FPGA vendors have recently introduced hybrid platforms consisting of high performance processors coupled with programmable logic, aimed at system-on-chip applications. These architectures partition the hardware into a processor system (PS), containing one or more processors along with peripherals, bus and memory interfaces, and other infrastructure, and the programmable logic (PL) where custom hardware can be implemented. The two parts are coupled together with high throughput interconnect to maximise bandwidth (Fig. 1). In this paper, we focus on the Xilinx Zynq-7000.

3.1 The Zynq-7000 Processing System (PS)

The Zynq-7000 contains a dual-core ARM Cortex A9 processor equipped with a double-precision floating point unit, commonly used peripherals, a dedicated hard DMA controller (PS-DMA), L1 and L2 cache, on chip memory and external memory interfaces. It also contains several AXI based interfaces to the programmable logic (PL). Each

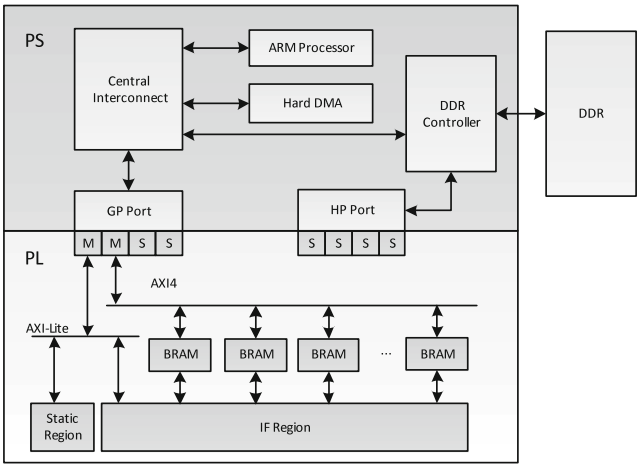

Figure 1 Block diagram of the hybrid platform.

interface consists of multiple AXI channels, enabling high throughput data transfer between the PS and the PL, thereby eliminating common performance bottlenecks for control, data, I/O, and memory. The AXI interfaces to the fabric include:

- AXI_ACP – One 64-bit AXI accelerator coherency port (ACP) slave interface for coherent access to CPU memory
- AXI_HP – four 64-bit/32-bit configurable, buffered AXI high performance (HP) slave interfaces with direct access to DDR and on chip memory
- AXI_GP – two 32-bit master and two 32-bit AXI general purpose (GP) slave interfaces

3.2 Programmable Logic (PL)

The Zynq-7000 PL is a Xilinx 7-series FPGA fabric comprising configurable logic blocks (CLB), Block RAM (BRAM), and high speed programmable DSP blocks. To make efficient use of the PL, we consider three possible uses, which may co-exist in various combinations. These are:

- **Static regions:** these consist of a fixed segment of the PL, defined at system configuration time, which contain pre-defined, frequently used accelerators, soft DMA controllers for data movement and state machines to control logic configuration.
- **Dynamic partially reconfigurable (DPR) regions:** used to implement less frequently used accelerators in a time multiplexed manner. Different accelerators can share DPR regions based on the availability of the region. However, this has a significant reconfiguration time overhead, as new bitstreams must be loaded. Reconfiguration time depends on the data size and speed of reconfiguration. While DPR allows for

the use of highly customised IP cores with maximum performance, if accelerators are regularly swapped into and out of the DPR region, reconfiguration time can adversely affect overall system performance.

We plan to use a DPR based reconfiguration scheme to reconfigure more highly optimized, but less frequently used, accelerators. The Zynq platform allows for two different reconfiguration ports, PCAP and ICAP. PCAP generally provides a sufficient, but not optimally fast solution. When reconfiguration speed is important we would use an ICAP solution, such as the high speed ICAP controller developed in [43].

– **Intermediate Fabric (IF) regions:** are built of an interconnected network of coarse-grained processing elements overlaid on top of the original FPGA fabric. The possible configuration space and reconfiguration data size is much smaller than for DPR because of the coarser granularity of the IF. An IF provides a leaner mechanism for hardware task management at runtime as there is no need to prepare distinct bitstreams in advance using vendor-specific compilation (synthesis, map, place and route) tools. Instead, the behaviour of the IF can be modified using software controlled configurations. However, mapping circuits to IFs is less efficient than using DPR or static implementations as the IF imposes a significant area and performance overhead.

3.3 Platform Communication Mechanisms

Instead of developing application-specific logic infrastructure to manage transportation of data to and from the PL, a memory abstraction can be used similar to the one described in [8]. This abstraction acts as a bridge between the PL and external memory interfaces. Dual-port BRAMs are mapped within the AXI memory mapped address domain, with one port connected to the AXI interface and the other to the PL. Using this abstraction allows the PS to communicate with BRAMs similar to the way it communicates with main memory. As mentioned earlier, the Zynq platform contains a number of high speed AXI interfaces which provide high bandwidth communication between the PL and main memory. To provide communication customisation and flexibility we allow three different communication mechanisms:

– **Non-DMA:** The CPU controls the flow of data between the main memory and the PL on a sample by sample basis via an AXI GP port. The maximum bandwidth for non-DMA communications, using a single 32-bit AXI GP port, is approximately 25 MB/s.
– **PS hard DMA:** The PS DMA controller takes small chunks of data from the main memory and sends them to the PL via an AXI GP master port. This dedicated hard DMA controller is able to perform memory to memory burst transactions and doesn't require any FPGA resources. As the GP master port width is 4 bytes and the maximum burst length is 16, a single burst it can transfer a maximum of 64 bytes. The CPU is able to perform other tasks during these transactions and can be interrupted by the DMA controller at the end of data transfer. Xilinx provides bare metal SW drivers for PS DMA.
– **Soft DMA:** Soft DMA uses an FPGA soft IP core to control the data movement between the memory and PL. Data is transferred between the main memory and PL via an AXI HP or AXI ACP port. Two variations are possible: memory mapped to memory mapped (MM-MM DMA) transactions and memory mapped to streaming (MM-S DMA) transactions. For soft DMA transfers the CPU is free during the transactions and can be interrupted by the DMA controller IP core at the end of data transfer. We have used the AXI Central DMA Controller IP core from Xilinx as a soft DMA for MM-MM transactions between DDR and Dual port BRAMs. The maximum burst length is 256, which means that with a burst size of 4 bytes (to match our BRAM data width) we can transfer 1K bytes in a single burst. Xilinx also provides bare metal SW drivers to use with their IP core. We present experimental Results in Section 6.1 to show the effect of the SW driver on the communication overhead.

4 Reconfiguration Management

Dynamic partial reconfiguration (DPR) has been proposed as a powerful technique to share and reuse limited reconfigurable resources. However, poor hardware and software support for DPR and large configuration file sizes make it difficult and inefficient. As discussed earlier, we support two mechanisms for runtime reconfiguration: DPR regions and IF regions. DPR based reconfiguration requires management and control associated with the loading of configuration bitstreams, while IF based reconfiguration simply requires the setting of configuration registers. Hence, IF based reconfiguration can be very fast, since the amount of configuration data is relatively short, although the IF logic abstraction does result in a performance overhead.

4.1 IF Overview

The IF overcomes the need for a full cycle through the vendor implementation tools, instead presenting a much simpler problem of programming an interconnected array of processing elements. Moreover, it is possible to map a

large application to multiple successive uses of the same IF (context frames), hence enabling applications with a larger resource requirement than that available on the device.

The IF architecture described in this paper is aimed at streaming signal processing circuits and consists of a data plane and a control plane as shown in Fig. 2. The data plane (shown in Fig. 3) contains programmable processing elements (PEs) and programmable interconnections, whose behaviour is defined by the contents of the configuration registers referred to as context frame registers. The control plane controls the movement of data between the data plane and the Block RAMs, and provides a streaming interface between them. The control plane consists of the context frame registers and a context sequencer (finite state machine) which is responsible for loading the context frame registers from the context frame buffer (CFB). The context frame buffer (CFB) can hold multiple sets of context frame registers for multi-context execution.

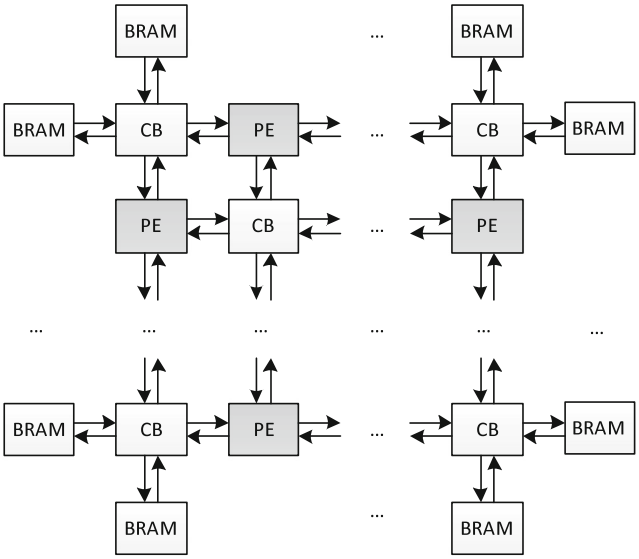

Figure 3 Block diagram of the data plane connected with abstracted memories.

4.2 Internal Architecture of the Data Plane

The data plane used in this paper consists of programmable PEs distributed across the fabric in a grid. DSP Blocks or CLBs can be used to implement these PEs depending on the required granularity. A PE is connected to all of its 8 immediate neighbours using programmable crossbar (CB) switches. We use 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 configuration registers, respectively.

Figure 4 shows the internal architecture of the DSP block based PE, which consists of a DSP block and a routing wrapper. The DSP block can be dynamically configured using the information contained in the PE configuration register. In our current implementation, we have used a fixed latency of 2 cycles for the DSP block. One of the key benefits of using DSP blocks in modern Xilinx devices is their dynamic programmability and wide range of possible configurations that can be set at runtime using control inputs. We have previously demonstrated their use as the base for a fully functional processor [7].

We currently perform manual mapping of HW tasks to the IF, as we are still developing an automated tool chain targeting our IF. However, a fully automated system could be implemented relatively easily by using an existing reconfigurable array with support for hardware virtualization, such as the one described in [31], which already has a toolset available for mapping HW tasks.

Figure 5 shows the configuration format of the PE and CB configuration registers. Each PE configuration requires three 32-bit registers, while each CB configuration requires two 32-bit registers. *pe_config* contains the alumode, inmode and opmode settings needed to configure

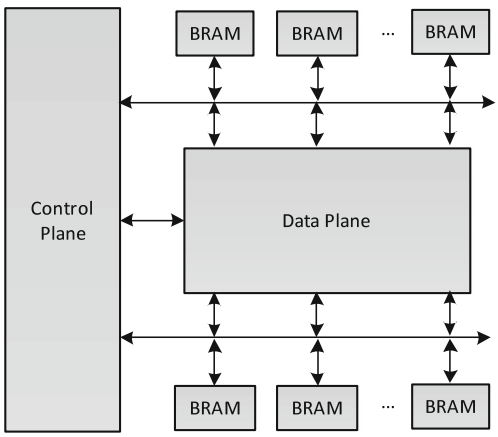

Figure 2 Block diagram of the intermediate fabric.

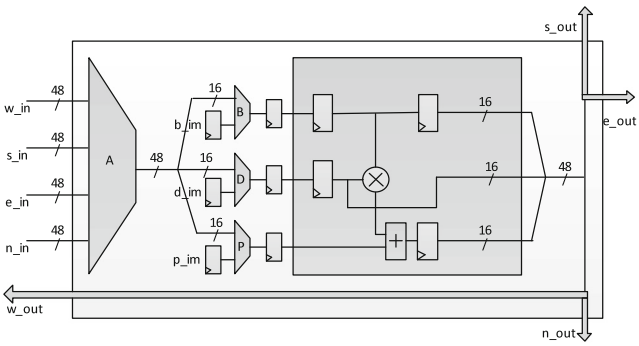

Figure 4 Internal architecture of processing element.

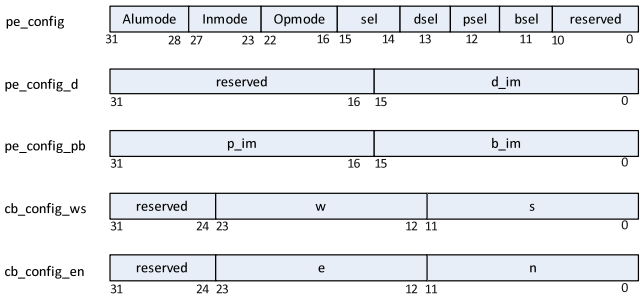

Figure 5 Configuration format of the intermediate fabric.

the DSP block function. *sel*, *dsel*, *psel* and *bsel* drive the select inputs of muxes A, D, P and B, respectively. Mux A determines the direction of the input data, while muxes D, P and B select between this input data or immediate data. Immediate data can be loaded before the start of computation using the contents of the *pe_config_d* and *pe_config_pb* configuration registers. The CB routing information is contained in *cb_config_ws* and *cb_config_en* configuration registers. The *w*, *s*, *e* and *n* fields contain the data connection for the west, south, east and north ports, respectively. As an example, a simple 32-bit IF containing 12 PEs and 13 CBs requires a 248 byte context frame and can be configured in 62 cycles. A 4KB Block RAM implementation of the CFB can hold 16 separate sets of context frame registers.

4.3 Internal Architecture of the Control Plane

The control plane used in this paper consists of a state machine based context sequencer and a context frame in the form of configuration registers. As mentioned in Section 4.1, a context sequencer (CS) is needed to load context frame registers into the configurable regions and to control and monitor the execution, including context switching and data-flow. We provide a memory mapped register interface (implemented in the PL fabric and accessible to the PS via the AXI bus) for this purpose. The control register is used by the PS to instruct the CS to start a HW task by checking the start bit of the control register. It contains the information about latency, number of samples to be processed and index of the input and output BRAMs for a particular task. 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 from the CFB to the control plane of the IF 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. 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 value of 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 equal to the context counter value, the whole HW task finishes and the CS moves to the *DONE* state. This behaviour is shown in Fig. 6.

5 The Hybrid Platform (Software)

While the integration of powerful multi-core processors with FPGA fabric (such as the Xilinx Zynq) has opened up many opportunities, designer productivity is likely to remain an issue into the near future. As mentioned in the Section 2, a number of researchers have developed techniques to provide OS support for FPGA based hybrid systems. However, as future high performance embedded systems are likely to include combinations of hard real-time, soft real-time and non real-time applications, all competing for compute capacity on a hybrid platform, a conventional OS may not provide the best solution. In this situation, an embedded hypervisor may be a better choice.

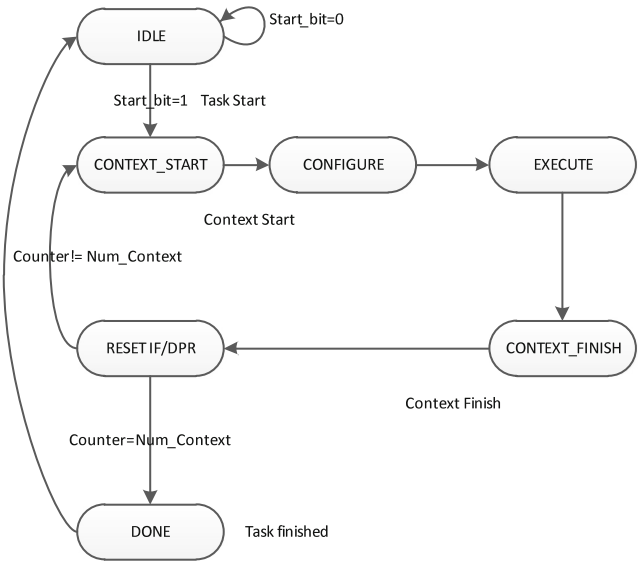

Figure 6 State-machine based Context Sequencer.

There are several existing embedded hypervisors or virtual machine managers in the market, some of which are certified for hard real-time systems. However, these hypervisors only support the PS and do not support the PL in a hybrid platform. Thus, PL virtualization and its integration into a hypervisor is important for virtualized execution of applications on hybrid computing platforms.

To address this problem, we have developed a general framework for a microkernel based hypervisor to virtualize the Xilinx Zynq hybrid computing platform so as to provide an abstraction layer to the user. The CODEZERO hypervisor [1] is modified to virtualize both hardware and software components of this platform enabling the use of the PS for software tasks and the PL for hardware tasks in a relatively easy and efficient way. The basic functionalities that the hypervisor is expected to perform on a hybrid platform are:

- Abstraction of resources and reconfiguration of the PL
- Communication and synchronization between PS and PL
- Scheduling of overlapping resources within and across tasks
- Protection of individual PS and PL tasks

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, support for hardware tasks 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. The hypervisor is able to dynamically modify the behaviour of the PL and carry out hardware and software task management, task-scheduling and context-switching, thus allowing time multiplexed execution of software and hardware tasks concurrently.

5.1 Porting CODEZERO to the Xilinx Zynq-7000

In this section, we describe the necessary modifications to the CODEZERO hypervisor [1], 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 PL virtualization. Currently, CODEZERO only runs on a limited number of ARM-based processors, and so we ported it to the Xilinx Zynq-7000 hybrid computing platform. The main changes included:

- Rewriting the drivers (e.g., PCAP, timer, interrupt controller, UART, etc.) for the Zynq-7000 specific ARM Cortex-A9 implementation
- Enabling MMU, Secondry CPU wake-up and PL initialization (e.g., FPGA clock frequency, I/O pin mapping, FPGA interrupt initialization, etc.)
- HW task management and scheduling

The UART, timer and interrupt controller are all essential for CODEZERO's operation, but are not specifically related to the requirements needed to support hardware virtualization. The UART is used for console display and debug, the timer is used by the CODEZERO scheduler and the generic interrupt controller (GIC) is for hardware management. The required changes to these routines were relatively trivial. For example, the interrupt controller only required a change in the base address, while the UART required a change in the base address as well as modifications to the initializing procedure due to changes in the register organization on the Zynq platform. The timer is essential for the correct operation of the CODEZERO scheduler.

Routines were also developed to enable the MMU and to wake the up secondary CPU in such a way that the application first initializes the secondary core's peripherals and then enables the interrupt sources. Initialization of the clock source and the clock frequency for the programmable logic is specific to the Zynq platform, and is not relevant for ARM only platforms.

5.2 HW Task Management and Scheduling

Hardware task management and scheduling is necessary to support hardware virtualization. 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.

5.2.1 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 an IF or DPR region in the PL so that other contexts cannot use the same region. This denotes the reconfigurable region 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 region 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

Table 1 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.

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 1.

5.2.2 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. The implementation of preemptive context switching is currently work in progress.

6 Experiments

In this section, we present details of the overheads associated with using a fully functioning virtualized hybrid system with a simple IF operating under CODEZERO hypervisor control. The IF consists of 12 DSP block based PEs, 13 CBs and 8 abstracted BRAMs. We present three separate examples using relatively small signal processing examples to demonstrate the overheads associated with our hypervisor based hybrid system. The first examines the communication overheads between IF and memory. The second examines the lock and context switch overheads associated with running a HW task under hypervisor control. The third experiment compares the hypervisor overhead to that of a more conventional implementation using an OS modified to support hardware tasks.

6.1 Communication Overheads

To examine the communication overheads, we use three simple 4, 8 and 12 tap FIR filter implementations which we have mapped to the IF. These small examples were chosen as they better demonstrate the overheads of the hybrid system, without task execution time dominating. It is possible to expand these tasks to more practical, larger ones with minimal effort. The operating frequency of the PS and the IF is 667 MHz and 100 MHz, respectively.

The PS initially loads the configuration data into the context frame buffer (CFB), sends the input data from main memory to the abstracted BRAMs and triggers the start of execution via the control register. Once the Context sequencer (described in Section 4) detects that the *start* bit in the control register has gone high, it configures the IF and starts streaming the data from the BRAM to the IF. The IF processes the data, sends the results to the destination BRAM and sets the *done* bit in the status register. These individual components are shown in Fig. 7, where the control register and status register access times are assumed to be negligible compared to the other components and are thus ignored. The task completion time T_{task} is essentially the sum of all the individual components shown in Fig. 7.

To further reduce the communication overhead, we overlap the loading of data to the BRAMs with the reading of the result data from the BRAMs using the multi-channel data transfer capability of hard-DMA. Thus the task completion time T_{task} can be reduced to:

$$T_{task} = T_{config-load} + T_{data-load} + T_{config} + T_{compute} \quad (1)$$

Non-DMA, hard-DMA and soft-DMA have all been used for data transfer as discussed in Section 3. In all cases, the

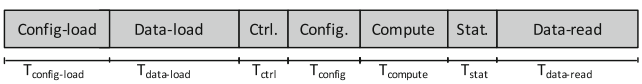

Figure 7 HW Task profile showing components effecting system performance.

Table 2 T_{task} in μs for Non-DMA based, Hard DMA, Soft DMA based and ARM only implementation of FIR filters.

Number of Samples	4-tap				8-tap				12-tap			
	ARM	Non DMA	Hard DMA	Soft DMA	ARM	Non DMA	Hard DMA	Soft DMA	ARM	Non DMA	Hard DMA	Soft DMA
64	19.62	17.05	24.4	26.42	36.73	20.10	27.56	29.58	55.38	23.87	30.72	32.74
128	38.98	29.53	28.55	29.81	73.01	32.26	31.71	32.97	109.92	36.23	34.87	36.13
256	77.41	54.50	37.58	36.45	145.67	56.57	40.74	39.61	219.32	61.18	43.9	42.77
512	153.32	101.85	55.56	46.48	292.83	105.19	58.72	49.64	438.27	108.69	61.88	52.8
1024	307.56	198.57	89.33	65.05	587.27	199.89	92.49	68.21	872.10	205.40	95.65	71.37

operating frequency of the data communication channel is 100 MHz. The data width of the channel is set to 4 bytes to match the data width of our dual port BRAMs. The communications overheads (T_{task}) for non-DMA based, hard-DMA based, soft-DMA based and the ARM processor only implementation for the three filters are shown in Table 2. The times are all in μs.

The non-DMA, hard-DMA and soft-DMA results from Table 2 are split into three parts, as $T_{config-load}$, $T_{data-load}$ and $T_{config} + T_{compute}$, to show the contribution of the individual components, as shown in Figs. 8, 9 and 10.

It is clear from Fig. 8 that for this simple application, the compute time represents only 5 % of the total time. The time required for data transfer from main-memory to the abstracted BRAM ($T_{data-load}$) is the main limiter of performance. The performance of the hard-DMA and soft-DMA based implementations are almost the same (approximately 3-4× reduction in data communication overhead compared to the non-DMA case, as shown in Figs. 9 and 10). This is because of the overheads associated with the interrupt based bare metal IP core SW drivers provided by Xilinx for the small data transfers used in the experiments. Reduce the entire IF can be reconfigured in 62 cycles, as discussed in Section 4, which for an IF frequency of 100MHz corresponds to a reconfiguration time of 620 ns. This time is lower for the smaller filters as we are able to partially reconfigure the IF for the smaller FIR filters. The configuration data and the configuration time for the FIR filter applications are shown in Table 3.

It should be noted that the time to load the configuration from main-memory to the CFB ($T_{config-load}$) does not change between non-DMA and DMA implementations. This is because we do not use DMA transfer to load the configuration, as the DMA overhead for the small amount of configuration data would make this inefficient. Although, DMA based configuration transfers may be advantageous if the IF size is increased significantly. The configuration overhead could be further reduced by implementing a configuration caching mechanism in CFB.

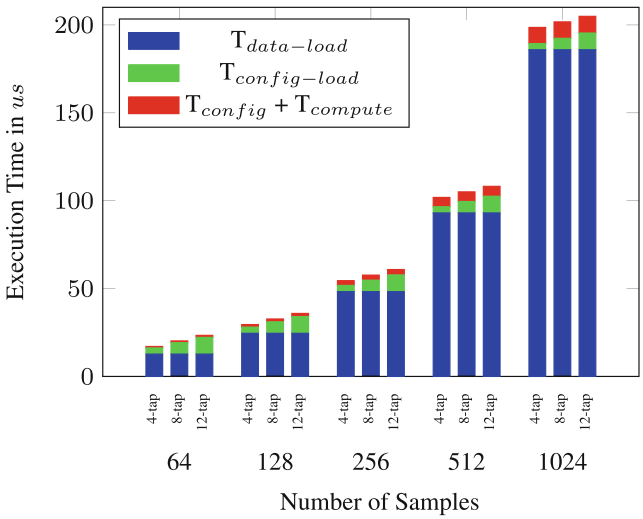

Figure 8 Individual component time for Non DMA based implementation.

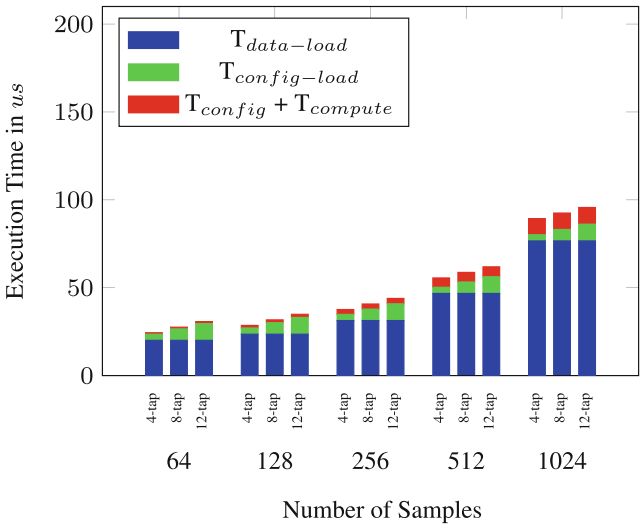

Figure 9 Individual component time for Hard-DMA based implementation.

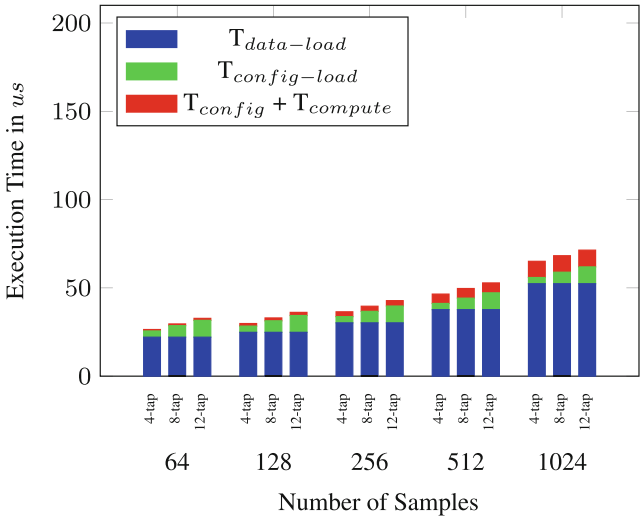

Figure 10 Individual component time for Soft-DMA based implementation.

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
```

6.2 Context Switch Overhead

The second experiment measures the overhead associated with CODEZERO's lock and context switch mechanisms when using the IF to run hardware tasks. We run two independent hardware tasks in two CODEZERO containers. The first container (cont0) is a bare metal application (an application which directly accesses the hypervisor APIs and does not use a host OS) which runs a 4-tap FIR filter as a hardware task on the IF. This filter is used to process 32 samples which takes 11.2μ s for task completion and 3.2μ s for loading the configuration. The second container (cont1) is also a bare metal application which runs a simple hardware matrix multiplication (MM) task on the same IF. We use a non-DMA transfer via the AXI GP port. As the two hardware tasks are executed on the same IF, scheduled and isolated by the hypervisor, 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.

To measure the hardware task activity we use a 64-bit global timer running at 333 MHz. We then measure every lock, unlock, IF configuration, IF execution and context switch activity which occurs when the two hardware tasks are running. The two containers are then mapped to the hybrid platform using two different scenarios: In the first, the two containers run on the same physical CPU core, while in the second, the two containers run on separate CPU cores. These two mapping scenarios are illustrated in Fig. 11.

In the first scenario, cont0 and cont1 are mapped to the same core, Core0, and the l4_context_switch is used to switch between cont0 and cont1 (the hardware contexts run one after the other without lock contention). When a context of FIR is finished, CODEZERO switches to MM, and vice versa. With this scenario, the lock overhead does not contain any of the working time of the other hardware task, as they occur sequentially.

In the second scenario, cont0 is mapped on Core0 while cont1 is mapped to Core1. As the two containers run on individual cores simultaneously both containers try to access the IF. In this case, as we operate in non-preemptive mode, the lock overhead associated with a hardware task contains some of the working time of the other hardware task as it must wait until the resource is free before it can obtain it, as illustrated in Fig. 12

Table 3 Reconfiguration time and configuration data size for FIR filters.

	Reconfiguration time	Configuration Data Size
4-tap FIR	220 *ns*	88 Bytes
8-tap FIR	420 *ns*	168 Bytes
12-tap FIR	620 *ns*	248 Bytes

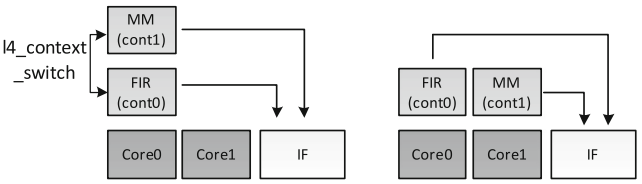

Figure 11 HW tasks on one core (Scenario 1) and on separated cores (Scenario 2).

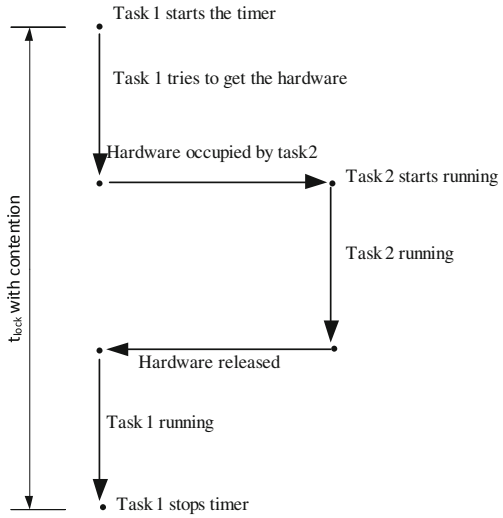

Figure 12 Lock with contention.

The hardware response time using a non-preemptive context switch is calculated as:

$$T_{hw\_resp} = T_{lock} + T_{C0\_switch} + T_{config-load} \quad (2)$$

the corresponding hardware response time using a preemptive context switch can be calculated as:

$$T_{hw\_resp} = T_{C0\_switch} + T_{read-state} + T_{config-load} \quad (3)$$

where $T_{config-load}$ is the time taken to load a new application configuration (HW task) into the IF and depends on the IF size and the data transfer rate, while $T_{read-state}$ is the time taken to read back the complete internal state (but not the configuration) of the IF. $T_{read-state}$ depends on the IF architecture, the IF size and the data transfer rate. For the IF with a PE internal architecture shown in Fig. 4, we need to save the internal state of the 7 registers to the right of the three muxes labelled B, D and P. This represents 28 bytes, compared to the 20 bytes of configuration data, as shown in Fig. 5. Thus, we can expect that $T_{read-state}$ is approximately 140 % of $T_{config-load}$.

The CODEZERO average lock and context switch overhead are shown in Table 4, while the configuration and response times are given in Table 5. For a single context switch on Linux, an overhead of 48μs was measured in [10] which is significantly less than the overhead for CODEZERO.

Table 4 Non-preemptive Lock and Context Switch Overhead.

	Clock cycles (time)
T_{lock} (no contention)	214 (0.32μ s)
T_{lock} (with contention)	7738 (11.6μ s)
T_{C0_switch}	3264 (4.9μ s)

Table 5 Configuration, Execution and Response Time

	Clock cycles (time)	
	FIR	MM
$T_{config-load}$	2150 (3.2μs)	3144 (4.7μ s)
T_{hw_resp}	(8.5μs-19.7μs)	(9.9μs-20.3μ s)

The lock time with no contention, which occurs when a task directly obtains a lock without needing to wait for the completion of another task, and the context switch time represent the true overhead. The other three times (lock with contention, configuration and HW response) vary as both the configuration overhead and the application execution time are affected by the data throughput, application complexity, and IF size.

To demonstrate that the processor is able to be usefully occupied while the HW tasks are executing, we execute two HW tasks in separate hypervisor containers, along with a simple software RTOS (μC/OS-II) running 14 independent tasks in a third container. All of the SW and HW tasks are scheduled and isolated by the hypervisor (as shown in Fig. 13) without affecting the HW response time (despite the extra container executing independent SW tasks, running under control of μC/OS-II).

This is because the HW response time depends on the context switch overhead, the lock overhead and the configuration load overhead, as shown in Eq. 2. The first two overheads are independent of the number of tasks being executed, while the last depends on the transfer speed to the IF. Additionally, the hypervisor schedules CPU resource based on containers, not tasks, and all will have an equal chance of being scheduled to run (assuming that they all have same priority). Thus, while individual tasks running under RTOS control will need to compete for CPU resource, the two hardware tasks will be largely unaffected.

6.3 Idle Time Comparison of CODEZERO and Linux

We have earlier made the assertion that a modified embedded hypervisor may be better than a modified OS for

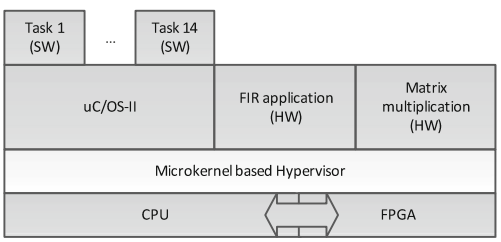

Figure 13 Multiple SW and HW task scheduling.

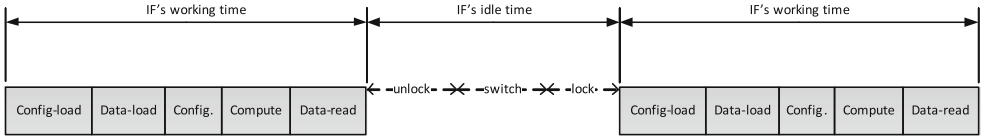

Figure 14 Idle time of IF between two consecutive tasks.

controlling hardware tasks running on the PL. Here, we perform a simple experiment to compare the overheads of CODEZERO and our modified version of Embedded Linux (kernel version 3.6) when running hardware tasks. To determine the overhead, we measured the idle time of the IF between two consecutive hardware contexts using a hardware counter instantiated in the FPGA fabric. A shorter idle time means a shorter (and thus better) context switch overhead in kernel space. Two hardware tasks are run repeatedly without any interrupt or preemption. In the Linux scenario, other system processes are running and compete for CPU time while the HW tasks are running.

A hardware context has the following steps (as described in Algorithm 1) to complete its execution: firstly the IF is locked, the configuration and input data are transferred, the execution cycle is entered and the results are generated, The results are read from the IF, lastly, the IF is unlocked making it available the next hardware context. The idle time between any of two consecutive hardware contexts can be defined and illustrated as in Fig. 14. The hardware counter will start counting after the results are read at the end of task execution and then stops counting when the first configuration data for the next hardware task arrives.

The same two hardware tasks (the FIR and MM implementations described in Section 6.2) are implemented in separate containers on CODEZERO, mapped to the CPU cores using the same scenarios as in Fig. 11. Therefore, in the first scenario, the idle time is caused by the lock overhead and the CODEZERO context switch overhead, while the idle time of the IF in the second scenario consists of the pure lock overhead, exclusive of any context switch overhead. The same 2 tasks are implemented in Linux with the *pthreads* library and mutex lock. Here, task switching and core allocation is controlled by the Linux kernel scheduler, and thus either task can run on either core dynamically and transparently.

The hardware idle time results for CODEZERO and Embedded Linux are shown in Table 6, and show that the hardware task overheads for CODEZERO are two orders of magnitude better than that of Embedded Linux. In CODEZERO, the idle time varies from 0.32μs to 5.4μs. The best case (0.32μs) occurs when the two containers run on separate cores, competing to obtain the IF lock. Thus, only the lock overhead, without any context switch overhead, is measured. The worst case (5.4μs) occurs when the two containers run on the same core, and thus the idle time includes both the lock and context switch overheads.

In Linux, the idle time varies from 43.17μs to 149.46μs. This wide variation occurs because Linux is not as lightweight as CODEZERO and has additional background tasks and unknown system calls running which will seriously affect the IF idle time.

Table 6 IDLE time on CODEZERO and embedded Linux.

CODEZERO	Embedded Linux
0.32μs-5.4μs	43.17μs-149.46μs

7 Conclusions and Future Work

We have presented a framework for hypervisor based virtualization of both hardware and software tasks on hybrid computing architectures, such as the Xilinx Zynq. The framework accommodates execution of software tasks on the processors, 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 hardware tasks in the FPGA fabric, again as either bare-metal hardware applications or as hardware-software 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.

We presented experiments to quantify the communication overhead between the processor system and programmable logic, demonstrating the key role this plays in determining the performance of such a software-hardware system. We also characterised the context switch overhead of the hypervisor approach, and compared the resulting overhead to a modified Linux approach, showing two orders of magnitude improvement in fabric idle time.

We are working on support for partial reconfiguration, with faster configuration through 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.

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

References

1. Codezero project overview. http://dev.b-labs.com/.
2. Adams, K. (2006). 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*.
3. Adler, M., Fleming, K.E., Parashar, A., Pellauer, M., Emer, J. (2011). Leap scratchpads: automatic memory and cache management for reconfigurable logic. In *Proceedings of International Symposium on Field Programmable Gate Arrays (FPGA)* (pp. 25–28).
4. Brebner, G. (1996). A virtual hardware operating system for the Xilinx XC6200. In *Proceedings of International Workshop on Field-Programmable Logic and Applications (FPL)* (pp. 327–336).
5. Callahan, T., Hauser, J., Wawrzynek, J. (2000). The Garp architecture and C compiler. *Computer*, *33*(4), 62–69.
6. Chakraborty, S., Lukasiewycz, M., Buckl, C., Fahmy, S.A., Chang, N., Park, S., Kim, Y., Leteinturier, P., Adlkofer, H. (2012). Embedded systems and software challenges in electric vehicles. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)* (pp. 424–429).
7. Cheah, H.Y., Fahmy, S.A., Maskell, D.L. (2012). iDEA: A DSP block based FPGA soft processor. In *Proceedings of International Conference on Field Programmable Technology (FPT)* (pp. 151–158).
8. Chung, E.S., Hoe, J.C., Mai, K. (2011). CoRAM: an in-fabric memory architecture for FPGA-based computing. In *Proceedings of International Symposium on Field Programmable Gate Arrays (FPGA)* (pp. 97–106).
9. Compton, K., & Hauck, S. (2002). Reconfigurable computing: a survey of systems and software. *ACM Computing Survey*, *34*(2), 171–210.
10. David, F.M., Carlyle, J.C., Campbell, R.H. (2007). Context switch overheads for Linux on ARM platforms. In *Proceedings of Workshop on Experimental Computer Science* (p. 3).
11. DeHon, A. (1996). DPGA utilization and application. In *Proceedings of International Symposium on Field Programmable Gate Arrays (FPGA)* (pp. 115–121).
12. DeHon, A., Markovsky, Y., Caspi, E., Chu, M., Huang, R., Perissakis, S., Pozzi, L., Yeh, J., Wawrzynek, J. (2006). Stream computations organized for reconfigurable execution. *Microprocessors and Microsystems*, *30*(6), 334–354.
13. Gohringer, D., Werner, S., Hubner, M., Becker, J. (2011). RAMPSoCVM: runtime support and hardware virtualization for a runtime adaptive MPSoC. In *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)* (pp. 181–184).
14. Gonzalez, I., & Lopez-Buedo, S. (2012). Virtualization of reconfigurable coprocessors in HPRC systems with multicore architecture. *Journal of Systems Architecture*, *58*(6), 247–256.
15. Heiser, G., & Leslie, B. (2010). The OKL4 microvisor: convergence point of microkernels and hypervisors. In *Proceedings of ACM Asia Pacific Workshop on Systems* (pp. 19–24).
16. Heiser, G., Uhlig, V., LeVasseur, J. (2006). Are virtual-machine monitors microkernels done right. *ACM SIGOPS Operating Systems Review*, *40*(1), 95–99.
17. Hubner, M., Figuli, P., Girardey, R., Soudris, D., Siozios, K., Becker, J. (2011). A heterogeneous multicore system on chip with run-time reconfigurable virtual FPGA architecture. In *IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)* (pp. 143–149).
18. Hubner, M., Gohringer, D., Noguera, J., Becker, J. (2010). Fast dynamic and partial reconfiguration data path with low hardware overhead on Xilinx FPGAs. In *IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW)* (pp. 1–8).
19. Jozwik, K., Tomiyama, H., Honda, S., Takada, H. (2010). A novel mechanism for effective hardware task preemption in dynamically reconfigurable systems. In *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)* (pp. 352–355).
20. Kaiser, R., & Wagner, S. (2007). Evolution of the PikeOS microkernel. In *Proceedings of International Workshop on Microkernels for Embedded Systems* (p. 50).
21. Kalte, H., & Porrmann, M. (2005). Context saving and restoring for multitasking in reconfigurable systems. In *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)* (pp. 223–228).
22. Kapre, N., Mehta, N., deLorimier, M., Rubin, R., Barnor, H., Wilson, M., Wrighton, M., DeHon, A. (2006). Packet switched vs. time multiplexed FPGA overlay networks. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (pp. 205–216).
23. Kirchgessner, R., Stitt, G., George, A., Lam, H. (2012). VirtualRC: a virtual FPGA platform for applications and tools portability. In *Proceedings of International Symposium on Field Programmable Gate Arrays (FPGA)* (pp. 205–208).
24. Kosciuszkiewicz, K., Morgan, F., Kepa, K. (2007). Run-time management of reconfigurable hardware tasks using embedded linux. In *Proceedings of International Conference on Field Programmable Technology (FPT)* (pp. 209–215).
25. Lavin, C., Padilla, M., Lamprecht, J., Lundrigan, P., Nelson, B., Hutchings, B. (2011). HMFlow: accelerating FPGA compilation with hard macros for rapid prototyping. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.
26. Liang, Y., Rupnow, K., Li, Y., et al. (2012). High-level synthesis: productivity, performance, and software constraints. *Journal of Electrical and Computer Engineering*, *2012*(1), 1–14.
27. Liedtke, J. (1995). On micro-kernel construction. In *Proceedings of the ACM Symposium on Operating Systems Principles* (pp. 237–250).
28. Lübbers, E., & Platzner, M. (2009). ReconOS: multithreaded programming for reconfigurable computers. *ACM Transactions on Embedded Computing Systems*, *9*(1), 8.
29. Mishra, M., & Goldstein, S. (2007). Virtualization on the tartan reconfigurable architecture. In *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)* (pp. 323–330).
30. Pham, K.D., Jain, A., Cui, J., Fahmy, S., Maskell, D. (2013). Microkernel hypervisor for a hybrid ARM-FPGA platform. In *International Conference on Application-Specific Systems, Architectures and Processors (ASAP)* (pp. 219–226).
31. Plessl, C., & Platzner, M. (2005). Zippy - a coarse-grained reconfigurable array with support for hardware virtualization. In

IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP) (pp. 213–218).
32. Rupnow, K. (2009). Operating system management of reconfigurable hardware computing systems. In *Proceedings of International Conference on Field-Programmable Technology (FPT)* (pp. 477–478).
33. Rupnow, K., Fu, W., Compton, K. (2009). Block, drop or roll(back): Alternative preemption methods for RH multi-tasking. In *IEEE Symposium on Field Programmable Custom Computing Machines* (pp. 63–70).
34. Shreejith, S., Fahmy, S.A., Lukasiewycz, M. (2013). Reconfigurable computing in next-generation automotive networks. *IEEE Embedded Systems Letters*, *5*(1), 12–15.
35. Shukla, S., Bergmann, N.W., Becker, J. (2006). QUKU: a coarse grained paradigm for FPGA. In *Proceedings of Dagstuhl Seminar*.
36. So, H., Tkachenko, A., Brodersen, R. (2006). A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (pp. 259–264).
37. Steiger, C., Walder, H., Platzner, M. (2004). Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Transactions on Computers*, *53*(11), 1393–1407.
38. Stitt, G., & Coole, J. (2011). Intermediate fabrics: Virtual architectures for near-instant FPGA compilation. *IEEE Embedded Systems Letters*, *3*(3), 81–84.
39. Thomas, D., Coutinho, J., Luk, W. (2009). Reconfigurable computing: Productivity and performance. In *Asilomar Conference on Signals, Systems and Computers* (pp. 685–689).
40. Trimberger, S., Carberry, D., Johnson, A., Wong, J. (1997). A time-multiplexed FPGA. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)* (pp. 22–28).
41. Vahid, F., Stitt, G., Lysecky, R. (2008). Warp processing: Dynamic translation of binaries to FPGA circuits. *Computer*, *41*(7), 40–46.
42. Vipin, K., & Fahmy, S.A. (2012). Architecture-aware reconfiguration-centric floorplanning for partial reconfiguration. In *Proceedings of International Symposium on Applied Reconfigurable Computing (ARC)* (pp. 13–25).
43. Vipin, K., & Fahmy, S.A. (2012). A high speed open source controller for FPGA partial reconfiguration. In *Proceedings of International Conference on Field Programmable Technology (FPT)* (pp. 61–66).
44. Vipin, K., & Fahmy, S.A. (2013). Automated partitioning for partial reconfiguration design of adaptive systems. In *Proceedings of IEEE International Symposium on Parallel and Distributed Processing Workshops (IPDPSW) – Reconfigurable Architectures Workshop (RAW)*.
45. Vuletic, M., Righetti, L., Pozzi, L., Ienne, P. (2004). Operating system support for interface virtualisation of reconfigurable coprocessors. In *Design, Automation and Test in Europe (DATE)* (pp. 748–749).
46. Wang, W., Bolic, M., Parri, J. (2013). pvFPGA: accessing an FPGA-based hardware accelerator in a paravirtualized environment. In *Proceedings of International Conference on Hardware/Software Codesign and System Synthesis* (CODES+ISSS) (pp. 1–9).
47. Ye, Z., Moshovos, A., Hauck, S., Banerjee, P. (2000). CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proceedings of International Symposium on Computer Architecture (ISCA)* (pp. 225–235).

Abhishek Kumar Jain received the Bachelor degree in Electronics and Communication Engineering from Indian Institute of Information Technology, Allahabad in 2012. He worked as an intern at STMicroelectronics, India for his dissertation work. Prior to that he was awarded MITACS Globalink Scholarship to pursue research work at Electrical and Computer Engineering Department, University of Alberta, Canada in May 2011. Currently he is a PhD student in the School of Computer Engineering, Nanyang Technological University, Singapore. His research interests include computer architecture, reconfigurable computing, and virtualization of reconfigurable hardware in embedded systems.

Khoa Dang Pham received the Bachelor degree in Mechanical Engineering from Ho ChiMinh City University of Technology, Vietnam and the Master degree in Computer Engineering from Nanyang Technological University, Singapore, in 2008 and 2013, respectively. Currently he is an Embedded application/firmware engineer in Larion Computing Ltd., Vietnam. His engineering background ranges from automation control to FPGA design and kernel development.

Jin Cui received the Bachelor degree in Mechanical Engineering and Automation, and the Master degree in Computer Engineering from Northeastern University, China and the Ph.D. degree in the School of Computer Engineering, from Nanyang Technological University, Singapore, in 2004, 2007 and 2013, respectively. Currently he is working as a research associate in TUM CREATE ltd., Singapore. His research interests are in the areas of computer architecture, reconfigurable computing, power-thermal-aware optimization for multiprocessor, and real-time embedded systems.

Suhaib A. Fahmy received the M.Eng. degree in information systems engineering and the Ph.D. degree in electricaland electronic engineering from Imperial College London, UK, in 2003 and 2007, respectively. From 2007 to 2009, he was a Research Fellow with the University of Dublin, Trinity College, and a Visiting Research Engineer with Xilinx Research Labs, Dublin. Since 2009, he has been an Assistant Professor with the School of Computer Engineering, Nanyang Technological University, Singapore. His research interests include reconfigurable computing, high-level system design, andcomputational acceleration of complex algorithms. Dr Fahmy was a recipient of the Best Paper Award at the IEEE Conference on Field Programmable Technology in 2012, the IBM-Faculty Award in 2013, and is a senior member of the IEEE and ACM.

Douglas L. Maskell received the B.E. (Hons.), MEngSc and Ph.D. degrees in electronic and computer engineering from James Cook University (JCU), Townsville, Australia, in 1980, 1984, and 1996, respectively. He is currently an Associate Professor in the School of Computer Engineering, Nanyang Technological University, Singapore. He is an active member of the reconfigurable computing group in the Centre for High Performance Embedded Systems (CHiPES). His research interests are in the areas of embedded systems, reconfigurable computing, and algorithm acceleration for hybrid high performance computing (HPC) systems.