IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received June 15, 2019, accepted July 15, 2019, date of publication July 26, 2019, date of current version August 13, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2931475

Snap-On User-Space Manager for Dynamically Reconfigurable System-on-Chips

ANDREA GUERRIERI, (Member, IEEE), SAHAND KASHANI-AKHAVAN, MIKHAIL ASIATICI, AND PAOLO IENNE, (Senior Member, IEEE)
École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland
Corresponding author: Andrea Guerrieri (andrea.guerrieri@ieee.org)

ABSTRACT Due to increased embedded processing requirements, modern SoCs are becoming heterogeneous computing platforms by combining traditional processing units with custom reconfigurable hardware accelerators (HAs) on an FPGA fabric. However, efficiently managing such HAs in an embedded Linux environment involves handling Linux kernel source code and creating custom device drivers specific to a target platform, therefore negatively impacting development costs, portability and time-to-market. To address this issue, we present LEOSoC, a snap-on user-space manager for dynamically reconfigurable SoCs. Using LEOSoC does not require any specific version of the Linux kernel, nor to rebuild a custom driver for each new kernel release.LEOSoC consists of a base *hardware* system and a *software* layer which run on SoCs from various vendors. The system identifies the SoC on which it is running and auto-adapts its communication channels to the HAs accordingly. Furthermore, LEOSoC allows applications to partially or completely change the structure of the HAs at runtime without rebooting the system by leveraging the underlying SoC support for dynamic full/partial FPGA reconfiguration. The system has been tested on multiple commercial off the shelf (COTS) boards from different vendors, each one running different versions of Linux, therefore proving the real portability and usability of LEOSoC in a custom industrial design. Finally, we use a cloud detection algorithm for multispectral image processing as a showcase for LEOSoC's capabilities and performance.

INDEX TERMS Adaptive systems, field programmable gate array, multithreading, reconfigurable architectures, system-on-chip.

I. INTRODUCTION

The increased computational requirements exhibited by today's embedded systems have promoted the development of reconfigurable heterogeneous System-on-Chips: devices which integrate traditional hard IPs (embedded CPUs, memory controllers, communication interfaces) with a tightly-coupled FPGA fabric, as shown in Figure 1. Such tightly-integrated reconfigurable logic enables system developers to design custom Hardware Accelerators (HAs) as a function of their applications needs. Such increased device complexity comes at the expense of application programming effort, requiring improvements in existing tools focusing on usability [1]. One of the most effective software environments for managing these complex SoCs is an embedded Linux operating system. Indeed, 90% of today's embedded devices are based on Linux kernel derivatives [2]. The success of embedded Linux systems is largely attributed to the availability of a wide range of device drivers responsible for managing all the hard IPs in an SoC. These drivers are provided by SoC vendors and are eventually merged into the mainline Linux kernel repository. In fact, the total space contribution of device drivers in the Linux source code reached 57% of the total lines of code in version 4.6 of the kernel. In contrast, the kernel source code itself only comprised 1.2% of the total code contribution [2].

The associate editor coordinating the review of this manuscript and approving it for publication was Alba Amato.

A. MOTIVATION

User applications cannot directly communicate with hardware because that requires privileges such as executing

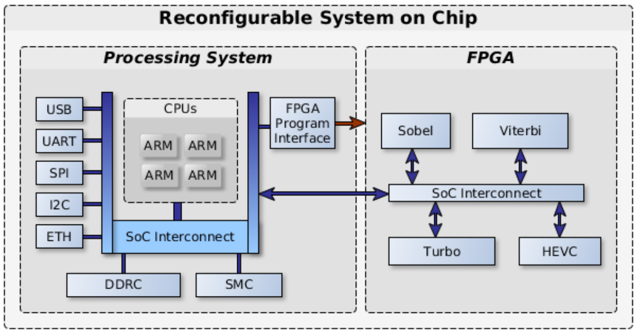

FIGURE 1. **Reconfigurable system-on-chip – internal structure.**

special instructions and handling interrupts. Device drivers assume the burden of interacting with hardware and export interfaces that applications and the rest of the kernel can use to safely access devices. Applications operate on devices via nodes in Linux's `/dev` directory and learn information about devices using nodes in the `/sys` directory [3]. Usually device drivers run in kernel space for protection reasons, with the drawback of being dependent on each version of the kernel.

While the Linux kernel shipped with SoC systems does include a set of precompiled drivers that enable software developers to rapidly develop reliable and ready-to-market applications using the *hard* peripherals of the SoC, nothing similar exists for the custom *soft* accelerators deployed on the FPGA. Indeed, an application developer who creates a custom HA on the FPGA must also write the custom driver that supports the accelerator and allows their applications to easily interface with it. Writing a custom driver for every HA is tedious for two reasons: (1) The driver would need to be adapted to each SoC platform to account for its specificities (different system topologies, peripherals, and addresses). (2) The driver would need to be *adapted* and recompiled whenever the Linux kernel is updated since the OS only accepts to dynamically insert a loadable device driver if it is compiled specifically for the *same* kernel version currently running on the platform. This makes sense since device drivers must match the kernel's driver interface. However, given that a new version of the kernel is released every 70 days [4] the portability of the device driver is severely restricted.

Furthermore, to decouple the Linux kernel developer teams from hardware vendors, the community introduced the concept of a *device tree*. By using a device tree, the description of the hardware structure (such the base address, the registers addresses, and the driver name of all hardware peripherals) can be loaded at boot time from a file instead of being hard-coded into the OS kernel [5]. Although loading the hardware description at boot time poses no issues on systems with fixed hardware, it makes it difficult to change the hardware at runtime by reprogramming the FPGA as doing so would require restarting the operating system to load a new device tree. To address this problem, has been introduced the concept of Device Tree Overlay (DOT), which allows to overwrite the device tree at runtime [6]. In fact, the newest Xilinx tool

```
#include <iostream>
#include "LEOSoC.h"

void main (int argc, char *argv[] )
{
  uint32_t data_input, data_output;

  data_input = get_input ();

  init ( FOUR_HA_STRUCTURE );

  //Write data into shared memory

  if ( create ( 1, HA_TYPE_A ) == RESP_OK )
  {
      write_reg ( 1, REG_INPUT, data_input );
      join ( 1 );
      data_output = read_reg ( 1, REG_OUTPUT );

      //Read computed data from shared memory
  }

  return;
}
```

Listing 1. **Sample example program.**

SDSoC [7] is able to produce the device tree required for the hardware designed, but still it needs to reboot the system after the FPGA reconfiguration, and it does not avoid to the user developer to handle the Linux sources. Another facility Linux provides for managing hardware devices is the user-space driver subsystem (UIO) which we attempted to use in our work since it provides the ability to handle interrupts [8]. However, its support is typically not compiled into the kernel by default.

B. GOAL

Our primary goal is to allow software experts to easily access and use hardware accelerators deployed on the FPGA fabric of such SoCs. Listing 1 shows a full program which can interact with HAs through LEOSoC. The code is simple, compact, and has no kernel version or compilation dependencies. A simple inclusion of the *LEOSoC.h* header file at the top of the file is enough to use the library. Calling the `init()` function and passing it the `FOUR_HA_STRUCTURE` parameter allows one to instantiate a static hardware structure which can host up to 4 HAs simultaneously onto the FPGA. Through the call `create(1, HA_TYPE_A)` we load a hardware accelerator of type A which is then instantiated into one of the available slots in the static FPGA structure programmed before. Sending commands/data to the HA is achieved through the `write_reg(1, REG_INPUT, data_input)` function which takes care of hiding all the complexity related to interfacing with the FPGA interface from a software application. This function can be used to pass to the HA the shared memory addresses to read/write data, avoiding data movement. The application then waits for the HA to terminate its internal operations by using the `join(1)` function and finally retrieves the computed data using the `read_reg(1, REG_OUTPUT)` function. This code can be compiled and executed as-is on different platforms, for example on Xilinx and Altera (now Intel) devices, without changing a single line in the application code. Section IV shows how LEOSoC achieves this transparency.

C. PAPER ORGANIZATION

This paper is organized as follows: Section 2 presents related work by underling their main differences with respect to our software library. Section 3 describes the characteristics of LEOSoC. Section 4 shows how LEOSoC is internally constructed as well as the hardware it supports. Section 5 showcases performance and a sample use case of the library.

II. RELATED WORK

Over the years much prior work has tried to simplify the management of hardware accelerators from a software perspective. LinROS [11] uses a Linux device driver that automatically manages the software and hardware of the reconfigurable SoC at runtime. However, it is an extension of a specialized version of Linux which limits its portability to other custom platforms with all the constraints related to driver development. The ReconOS operating system [9] offers unified services for functions executing in software and hardware, and a standardized interface for integrating custom hardware accelerators. ReconOS leverages the well-established multi-threading programming model and extends a host operating system with support for hardware threads. ReconOS is currently specialized for Xilinx SoCs, does not support dynamic full bitstream reconfiguration, and portability to different custom platforms is unsupported. R3TOS [14] is another real-time operating system capable of interfacing hardware and software systems. It is focused on hardware reliability by enabling the system to auto-detect hardware faults. However, it is not based on Linux and therefore reduces its applicability. SPREAD [12] is a framework able to manage hardware and software threads where a lightweight OS kernel has been used for extend control to hardware threads. Again, is another non-linux OS extension with portability and flexibilty limitations. FUSE [10] is a front-end for easy integration of hardware threads. It is composed of two parts: TLFC which runs in user-space and LLFC which runs in kernel-space. Since FUSE includes some parts running in kernel-space, it cannot be completely free of kernel dependencies like LEOSoC, which runs completely in user-space. As a consequence, while FUSE can, from a high-level architecture perspective, work on any Linux platform, the LLFC module may have to be adapted every time the kernel version is changed. CAP-OS [13] is a runtime system capable of controlling and scheduling the reconfiguration of hardware partitions following a preemptive-scheduling approach. In CAP-OS, Xilkernel is used as a lightweight OS. Again, is not Linux-based and not portable to different platforms. Mini-NOVA [15] is a lightweight ARM-based virtualization microkernel supporting dynamic partial reconfiguration. It runs on Zynq SoCs from Xilinx and is based on an RTOS. It doesn't have any support for full dynamic reconfiguration and the portability to other platforms is restricted to the native design. Furthermore, it is RTOS-based and not Linux-based. Hthreads [16] is a framework for managing hardware accelerators from software. It is able to manage hardware threads on an FPGA in a similar way as the operating system does for software threads by supporting semaphores and interrupts. However it is an extension of an RTOS and therefore suffers from portability and flexibility issues limited to the specific platform it was developed on.

A. DIFFERENCES OF OUR WORK

Our work is different from those presented above for several reasons. Firstly, it was specifically designed for portability on different platforms. Secondly, with respect to useability, the goal was to make the library simple to use and, more importantly, easy to integrate in a real industrial design. By avoiding any kernel dependencies, LEOSoC can be used on any kernel version without changes in the source code. Moreover, LEOSoC has been designed to use the same source code (without any modification) on different platforms by different vendors, since it recognizes the platform where it is running and auto-adapts the system calls and address map accordingly. Adding new platform to LEOSoC is much easier compared to other systems since it does not require to handle the Linux kernel's source code neither to recompile it from scratch. This reduce dramatically the effort, skills, and time needed for adding new platforms to the existing ones.

III. CROSS-PLATFORM EMBEDDED LINUX LIBRARY

The new approach proposed in this paper aims to overcome the high effort required to interface with FPGA-based hardware accelerators in an embedded Linux environment. We focus our attention on making this software *cross platform*, *easy to use*, while at the same time being *reliable*. In this section we present the characteristics of the library. The library consists of two distinct parts which work together. The first part of the library consists of a portable base *hardware* template to be instantiated on the FPGA and which contains all the infrastructure needed to support up to N hardware accelerators, akin to the "shell" in Microsoft Catapult [17]. We call this base hardware template the "static design". The second part of the library is a *software* module which leverages the infrastructure of the static design to provide software developers with easy access to their hardware accelerators and allows them to partially/fully reconfigure the FPGA in their target platform.

A. CROSS PLATFORM

All the architectural constructs are made using the HAL (Hardware Abstraction Layer) software model, allowing to execute the same source code on different platforms, independently on the underlying hardware [18]. At startup, the library tries to discover the platform on which it is running and, once determined, adapts its behavior and consequently the address mapping accordingly. The source code is written without any constraints related to the structure of a particular SoC which would restrict the applicability of the library.

TABLE 1. Comparison to related work.

Work	Linux OS	Thread Model	Reconfiguration Partial/Full	Kernel Free	Cross Platform
ReconOS [9]	Yes	Yes	Yes / No	No	No
FUSE [10]	Yes	Yes	No / No	No	No
LinROS [11]	Yes	Yes	Yes / No	No	No
SPREAD [12]	No	Yes	No / No	No	No
CAP-OS [13]	No	No	Yes / No	No	No
R3TOS [14]	No	No	Yes / No	No	No
Mini-NOVA [15]	No	No	Yes / No	No	No
HThreads [16]	No	Yes	No / No	No	No
LEOSoC	**Yes**	**Yes**	**Yes / Yes**	**Yes**	**Yes**

For every supported platform, the platform include a *header file* containing all the references for the specific platform. Furthermore, the concept of portability is not based around FPGAs from the same family or from the same vendor. At the time of writing of this paper, our library supports Intel and Xilinx SoCs, however we plan on adding upcoming support for Microsemi and recent NanoXplore FPGAs to maximize applicability.

B. EASY TO USE

The second important element where we focus our attention is the usability of the framework. Building and incorporating the library in an application is simple. The library is statically built using a traditional GCC makefile, one of the *de facto standard* in embedded system development environment. The makefile generates a static `.a` library file which can then easily be included in the custom makefile of a target application.

C. RELIABILITY

LEOSoC supports FPGA Dynamic Full Reconfiguration (DFR) and Dynamic Partial Reconfiguration (DPR). The main problem with software support for dynamic full reconfiguration and dynamic partial reconfiguration is related to the management of different hardware structures while reprogramming the FPGA. Indeed, trying to access an accelerator's memory-mapped registers when the accelerator is physically absent causes the system to hang and requires a reboot of the whole machine. In our library this problem has been avoided by checking the hardware status before accessing the bus. Another relevant problem is related to the glitches during the dynamic partial reconfiguration. To address this problem, we introduced hardware isolation, explained in Section IV-C, automatically managed by the library without any intervention needed from the user.

IV. LIBRARY STRUCTURE

The internal structure of the library is designed in order to hide all the hardware structures needed to support the HAs from the software developer's workload. Figure 2 shows the internal structure of the library. Basically, the software library is mainly composed of two parts: a front-end and a back-end.

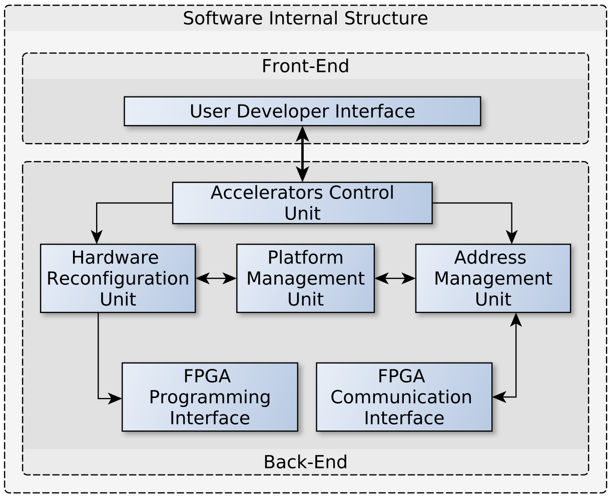
Software Internal Structure
Front-End
User Developer Interface
Accelerators Control Unit
Hardware Reconfiguration Unit
Platform Management Unit
Address Management Unit
FPGA Programming Interface
FPGA Communication Interface
Back-End

Figure 2. Library internal structure.

A. FRONT-END

User Developer Interface: The design of the front-end is inspired from the programming paradigm used to manage POSIX threads, inspired by HThreads [19]. This choice was mainly done from a user-friendliness principle as most embedded software developers are well-accustomed to this paradigm and they would not have to learn a completely different framework to be able to efficiently interface with their custom HAs. POSIX threads, PThreads [20] for short, is an execution model which support concurrent execution in a software environment. Similarly to the *virtual* software multi-threading that occurs in a classic CPU, an SoC with an integrated FPGA can support hardware threads in its FPGA to achieve *real* concurrent execution. In our work we have focused our attention on the basic management of HAs, but future implementations and optimizations to further match PThread-like semantics are planned. The complete list of public functions is reported in Table 2.

TABLE 2. List of public functions.

Method	Description
`init(type)`	Set the static design on the FPGA
`create(indx, type)`	Creates HA instance
`cancel(indx)`	Cancel HA instance
`join(indx)`	Wait specified HA has terminated
`destroy(void)`	Destroy the HA structure
`read_reg(indx, a)`	Read 32-bit value from HA
`write_reg(indx, a, val)`	Write 32-bit value to HA

The `create()` and `join()` concepts are familiar from the PThreads context. Additionally, we have introduced the `init()` and `destroy()` functions whose main scope is to define the *static design* of the FPGA, i.e., the FPGA structures needed to support a given number of HAs and their corresponding size in terms of device resources. Using these functions allows developers to allocate/deallocate a finite number of HAs at runtime on their target platform. Finally, the library provides a set of standard interface functions that grant developers access to their HA's internal registers so they can easily program their custom accelerators.

B. BACK-END

The back-end performs all the operations needed to interface the FPGA subsystem from software. It is composed of the following components:

- *Accelerator Control Unit;*
- *Address Management Unit;*
- *FPGA Communication Interface;*
- *Platform Management Unit;*
- *Hardware Reconfiguration Unit;*
- *FPGA Programming Interface.*

1) ACCELERATOR CONTROL UNIT

This unit represents the core of the library. The accelerator control unit manages and dispatches requests coming from the front-end to the FPGA subsystem for controlling, reprogramming and restructuring the HAs. It also manages the assignment of the HA into available partial-reconfigurable regions, in function of the availability of free HA slots. If the number of requests exceeds the available slots, it will be inserted in a queue, then dispatched when the slot become available.

2) ADDRESS MANAGEMENT UNIT

The user developers cannot directly access a physical address since the library provides an abstract access to the HAs. The address management unit is responsible for computing the physical address of the specific HA by evaluating the mapping function reported in equation [1]:

$$PhysAdd = BaseAdd_p + (HA_i \cdot HA_c). \quad (1)$$

where $BaseAdd_p$ represents the base address of the control bus specific to a given platform, HA_i is the hardware accelerator index provided by the user, and HA_c is a coefficient fixed by the hardware design currently loaded. Both $BaseAdd_p$ and HA_c are provided by the platform management unit, which stores information about the FPGA structure for the running session.

3) FPGA COMMUNICATION INTERFACE

From the operating system's perspective there cannot be any specific driver loaded for the HAs since one does not know what the structure of the FPGA for the running session is. Accessing the HAs registers through physical memory is instead managed by the library through the `/dev/mem` system device. In Linux, `/dev/mem` is a character device file that is an image of the physical memory of the system [21]. Indeed, using the `mmap` system call [22] it is possible to access the registers of the HAs present in the programmable logic of the FPGA at runtime, therefore avoiding any dependency on a Linux device tree or a Linux device driver. Furthermore, this approach enables the library to change the number and the size of the HAs at runtime, without reloading any device tree, driver or rebooting the system for using the new hardware peripherals.

4) PLATFORM MANAGEMENT UNIT

The library is capable of discovering its host platform. This discovery process is executed during the library's initialization phase. The `CHECK_FPGA()` function checks for the presence of the FPGA programming device in the Linux file system, which is distinct on every platform. Once a compatible programming device is found, the running environment can be set and consequently used from the address management unit for the computation of the physical addresses.

5) HARDWARE RECONFIGURATION UNIT

In reconfigurable SoCs, the processing system includes a special device that is responsible for reconfiguring the programmable logic on the FPGA fabric at runtime. Since this device is part of the hard IPs, a custom driver for this peripheral is already present in the Linux kernel, usually released with the reference Linux kernel image of the board. From the library's perspective, the reconfiguration is managed through the specific platform driver of the reconfiguration controller. The reconfiguration controller is visible to the operating system as a standard character *device file* and reconfiguration is done by simply writing the target bitstream into this device file.

6) FPGA PROGRAMMING INTERFACE

Each platform has its own custom *device file* for performing dynamic partial/full reconfiguration on the programmable logic. To detect which device file to use, the hardware reconfiguration unit queries the platform management unit for information about the platform's device at library initialization time. To provide a standard interface to this device file

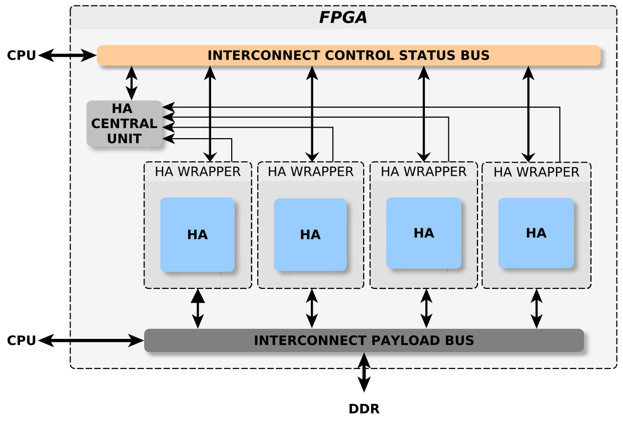

Figure 3. FPGA static design.

we created a virtual interface called FPGA programming interface which points to the correct device. After the first access to the device file, the hardware reconfiguration unit caches the device identifier in its environment variables for the current running session. The throughput of the reconfiguration controller depends on the capabilities of its actual hardware and on its device driver's capabilities. We show some reconfiguration throughput measurements in Section V.

C. BASE HARDWARE STRUCTURE

The base hardware structure is the improved version of the architecture presented by Guerrieri et al. [23] adapted to be synthesized on FPGAs from different vendors. This base hardware structure (i.e. *static design*) has been specifically designed to host multiple independent HAs with different purposes, sizes and functionalities. Figure 3 shows the internal structure of the static design infrastructure.

1) HARDWARE ACCELERATOR

Each HA shares the external memory with the CPU and with the other HAs through the Payload Bus. The HA can be automatically-generated using high-level synthesis tools such as Vivado HLS [24] or Intel HLS Compiler [25], or can be designed using the standard hardware description languages such as VHDL or Verilog. In order to normalize the HA's interfaces, to provide virtual memory protection, and to ensure the dynamic partial reconfiguration process, we introduced the hardware accelerator wrapper.

2) HARDWARE ACCELERATOR WRAPPER

We implement a standard *hardware accelerator wrapper* which we use to contain accelerators and control their access to the rest of the system. Having a standard wrapper also allows the system infrastructure to monitor and control the accelerator independently from the accelerator's functionality. Figure 4 shows the architecture of the hardware accelerator wrapper. It is it composed of (1) a wrapper control unit, (2) a partial-reconfigurable region (accelerator slot), (3) isolators, and (4) a Translation Lookaside Buffer (TLB). The wrapper control unit provides a set of

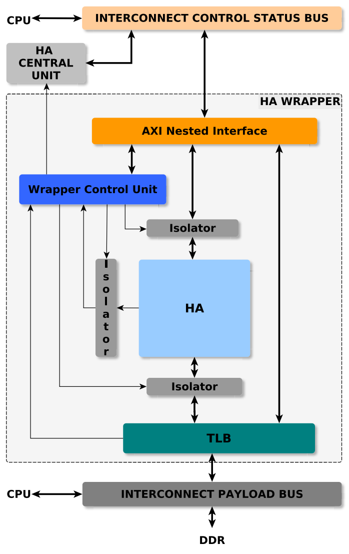

Figure 4. Hardware accelerator wrapper.

predefined registers for controlling the subsystem's components. An accelerator kernel can be loaded dynamically into each partial-reconfigurable region depending on user/system requests. Isolators provide a safe interface between the static design and dynamic design during partial reconfiguration, in order to avoid that hardware glitches affect the rest of the system during the reconfiguration process. Finally, a TLB is added to each hardware accelerator wrapper to enable memory protection and virtualization. If disabled, the TLB will transparently allow direct access to memory.

3) INTERNAL BUS

Controlling the HAs behavior from software requires to have a communication link between the CPU and the HAs. SoCs include several such interconnects which all use the AXI4 protocol, so all communication between the CPU and the HAs in the static design is AXI4-based. However, the HAs would achieve very low performance if they were to ask the host CPU to perform memory accesses for them as a middleman and communicate the result to them through a simple control/status bus. To get around this issue, we decouple the static design's communication infrastructure into two independent interconnects: the control/status bus and the payload bus. The control/status bus is an AXI4-Lite interconnect whose main role is to allow the host CPU to manage the HAs through their control/status interfaces by writing to control registers and reading from status registers. The payload bus is an AXI4-Full interconnect which gives accelerators high-performance direct access to system memory without involving the host CPU or the software library in any way.

4) HA CENTRAL UNIT

This unit is managed by the library and its goal is to intercept requests directed to the processor from the HAs, in order to centralize the requests. The library read the status of all the HAs through this unit using one bus transaction,

TABLE 3. Dynamic partial reconfiguration and dynamic full reconfiguration measurements for the selected target platforms.

	DPR			DFR		
Board	l_s (MB)	t_r (ms)	B_r (MB/s)	l_s (MB)	t_r (ms)	B_r (MB/s)
Xilinx	1.67	51	32.85	13.32	212	62.83
Altera	n.a.	n.a.	n.a.	4.24	325	13.02

independently of the number of HAs present in the static design. An example of such a request is the termination of a scheduled job or the notification of an error. The HA central unit is connected to the control/status bus and the number of HAs it supports is related to the number of HA slots present in the static design.

V. EVALUATION AND TESTS

A. EXPERIMENTAL SETUP

We tested the execution capability of the *same* library on *different* versions of Linux kernel without changing a single line in the source code. Furthermore, to demonstrate the real advantages coming from the uses of such as library for embedded systems, we developed and tested a realistic industrial application, a cloud detection algorithm for multi-spectral image processing. We deployed our system on two FPGAs from two major device vendors:

1) Xilinx ZC706 (Xilinx XC7Z045-FFG900-2, high-end)
2) Terasic DE0-Nano-SoC (Altera 5CSEMA4U23C6, low-end)

B. PERFORMANCE AND RESOURCE UTILIZATION

1) HARDWARE ACCELERATOR INSTANTIATION LATENCY

The `create()` function involves the dynamic partial reconfiguration. The reconfiguration time t_r is determined by two factors: the bandwidth of the reconfiguration controller B_r and the size of the bitstream l_s. However, the dynamic partial reconfiguration is performed only during the first instantiation of the HA, reducing the latency and improving the performance in running multiple times the same accelerator. Table 3 shows the measured reconfiguration time for dynamic full reconfiguration and dynamic partial reconfiguration, where supported by the underlying device.

2) READ/WRITE LATENCY

To measure the latency of our `read()` and `write()` APIs, we instantiate a simple accelerator with a single configuration register and we use Xilinx's Integrated Logic Analyzer to measure the pulse width on that register when toggling its value in a sequence of tight writes through our library. We compare this pulse width to the one observed when the same register is set by consecutive writes in a bare-metal application, which is an optimistic estimation of the latency we would measure if we were using a device driver. Table 4 shows that our user-space library only introduces a latency overhead of 55 *ns* on accessing the control/status interface of a given HA.

TABLE 4. Hardware access latency.

Board	Bare metal [ns]	Our library [ns]
Xilinx ZC706	170	225

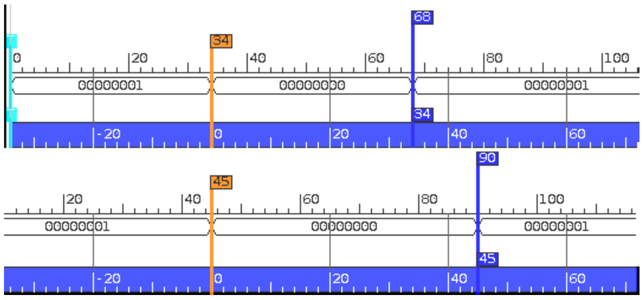

Figure 5. Hardware access latency from bare-metal software (top) and from user-space Linux software using LEOSoC (bottom), as measured by the Xilinx ILA (orange to blue cursor, bottom blue time scale in 200 MHz FPGA clock cycles). Even with our library, hardware access latency remains on the order of hundreds of nanoseconds just like bare-metal software, the latter providing an optimistic estimation of access times seen by a driver.

TABLE 5. Resource utilization of hardware system template. Note that Altera and Xilinx FPGAs have different units in their basic blocks, so no linear ratio can be derived between the utilization of slices vs. ALMs.

		Xilinx ZC706	Altera DE0-Nano-SoC
	PRRs	Slices [#]	ALMs [#]
Control Status IC	4	179	140
	8	324	255
HA Central Unit	4	37	25
	8	46	26
HA Wrappers	4	1857	3796
	8	3745	7603
Payload IC	4	272	261
	8	484	491
Total	4	2308	4222
	8	4553	8375

3) RESOURCE UTILIZATION

Table 5 shows the resource utilization of our hardware infrastructure on two different FPGAs. Computing area usage is tricky since it requires knowledge of what the basic building blocks of the specific FPGA used. In percentage, the static design occupies 4% of the FPGA resources of the Xilinx ZC706's SoC (high-end) while 28% on the Altera DE0-Nano-SoC's one (low-end).

C. BENCHMARK APPLICATION

1) CLOUD DETECTION ALGORITHM DESCRIPTION

The cloud detection algorithm is an application used in satellite for earth observation. The purpose is to process multi-spectral images to find clouds in images [26]. The analysis is performed applying a series of sequential filters, starting from the red region of the solar spectrum. The output is a probability of potentially cloud for each pixel. The algorithm fil-

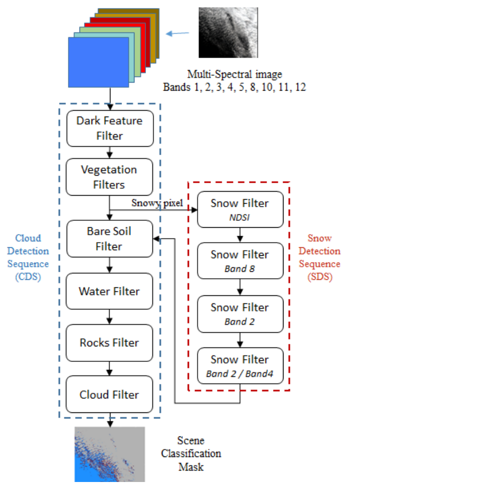

Figure 6. **Cloud detection algorithm steps. Algorithm input: Multi-spectral images. Algorithm output: Scene classification mask.**

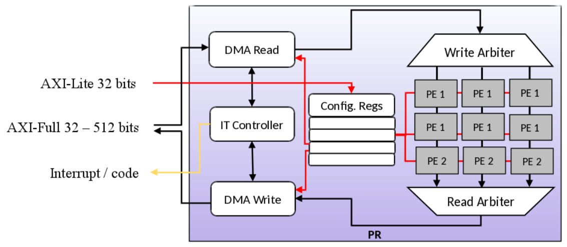

Figure 7. **System view of cloud detection hardware implementation. White boxes represent reusable modules. Gray boxes represent custom modules.**

tering sequence can be divided into two parts: the basic cloud detection sequence (CDS) and the snow detection sequence (SDS). The SDS is enabled if and only if the image contains snowy pixels. The SDS sequence may not be instantiated into the hardware thread that will detect clouds in snow-free regions to further save FPGA resources and power consumption. However, the snow sequence can be added whenever the image represents a snowy region thanks to the capability of our library. This can be simply done by swapping hardware accelerator implementing CDS and CDS + SDS sequences using the `create()` function call. Figure 6 shows the block diagram of the cloud detection algorithm used in our experiments.

2) HARDWARE ACCELERATOR IMPLEMENTATION

The algorithm's steps are implemented in a processing element (PE). The PE module may be instantiated as many times as the partial-reconfigurable region size allows it. Except for the PE modules, all the other modules of the hardware accelerator design are from the hardware accelerator template, such as DMA engines, AXI interfaces, configuration registers, arbiters, FIFOs and interrupt systems. Figure 7 shows the HA internal structure for the given application.

3) EXECUTION PERFORMANCE

We compiled and executed on the dual-core ARM Cortex-A9 processor the software version of cloud detection algorithm using 1, 2, and 4 software threads. Then, we compared the execution time using 1, 2, and 4 HAs. Figure 8 compares the execution time of the software-based algorithm to the various hardware-accelerated versions. Table 6 reports some time details on the application execution, showing the setup and the execution time for the different versions. This table shows the achievable speedup up to 34× in the hardware-accelerated version and how the HA's instantiation latency can affect the performance. In fact, although the HA execution time is smaller with multiple HAs, the total time increases. This is true processing a single image. In processing multiple images

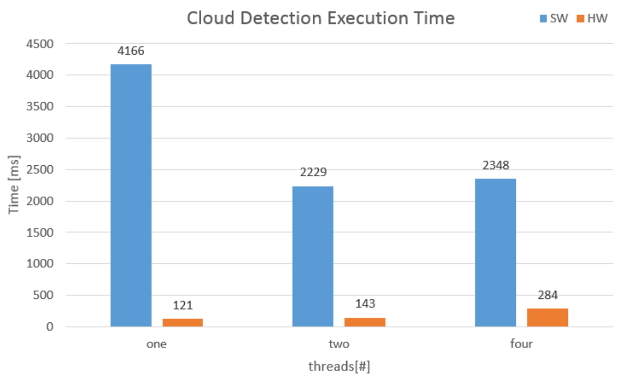

Figure 8. **Total execution time for various hardware configurations.**

TABLE 6. **Platform performance for various hardware configurations.**

Model	Setup Time [ms]	Execution Time [ms]	Total Time [ms]
SW-1	N.A	4166	4166
SW-2	N.A	2229	2229
SW-4	N.A	2348	2348
HW-1	63	58	121
HW-2	126	17	143
HW-4	252	32	284

TABLE 7. **Speedup processing multiple images for various hardware configurations.**

	Speedup			
Model	1 Image	5 Image	10 Image	100 Image
HW-1	34×	59×	65×	71×
HW-2	16×	53×	75×	122×
HW-4	8×	28×	41×	68×

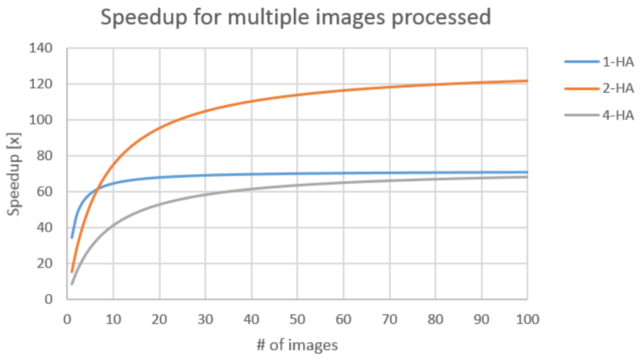

Figure 9. **Speedup For multiple image processed in sequence.**

in sequence the setup time disappears thanks to the caching feature of LEOSoC. Table 7 shows how the speedup change as a function of the number of images. Up to 5 images, the fastest version remains the one with one HA, while the version with 2 HAs becomes more effective starting from 10 images. Here is where LEOSoC play the key role: the user can decide at runtime how many HAs allocate and run concurrently to achieve the best performance, with a minimum design effort.

VI. CONCLUSION

This paper presents LEOSoC, a snap-on, user-space manager which solves the problem of managing hardware accelerators in dynamically-reconfigurable SoCs, with minimum development effort. As shown in the motivation and in the review of related work, most LEOSoC features are individually available in other solutions. Yet, we believe that LEOSoC represents a unique solution which combines pragmatically and effectively everything an embedded system developer needs to profit of emerging reconfigurable platforms with minimal risk and negligible effort. Besides, and contrary to some other academic embodiments of similar ideas, LEOSoC is designed with performance in mind and not to compromise the speedup one can achieve from hardware accelerators. We have shown that the overhead of accessing the HA registers from the CPU through our library is negligible compared to a bare-metal application and that realistic signal processing applications can achieve significant speedups with minimum costs. The source code of LEOSoC will be released as open-source available for download at *https://github.com/Andrea-Guerrieri/LEOSoC*.

REFERENCES

[1] *Versal: The First Adaptive Compute Acceleration Platform (ACAP)*,Xilinx, San Jose, CA, USA, 2018.
[2] *Linux Kernel and Driver Development Training*, Free Electrons, Paris, France, 2017.
[3] S. Venkateswaran, *Essential Linux Device Drivers*. Upper Saddle River, NJ, USA: Prentice Hall, 2008.
[4] *Status of Embedded Linux*, Linux Found. CE Workgroup, San Francisco, CA, USA, 2017.
[5] *Devicetree Specification*, DeviceTree.org, Cambridge, U.K., 2016.
[6] *Device Tree Overlay Notes*. Accessed: 2018. [Online]. Available: https://www.kernel.org/doc/Documentation/devicetree/overlay-notes.txt
[7] Xilinx. (2018). *SDSoC Environment Platform Development Guide*. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1146-sdsoc-platform-development.pdf
[8] (2006). *The Userspace I/O HOWTO*. [Online]. Available: https://www.kernel.org/doc/html/v4.18/driver-api/uio-howto.html
[9] A. Agne, M. Happe, A. Keller, E. Lübbers, B. Plattner, M. Platzner, and C. Plessl, "ReconOS: An operating system approach for reconfigurable computing," *IEEE Micro*, vol. 34, no. 1, pp. 60–71, Jan./Feb. 2014.
[10] A. Ismail and L. Shannon, "FUSE: Front-end user framework for O/S abstraction of hardware accelerators," in *Proc. IEEE 19th Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, May 2011, pp. 170–177.
[11] J. Rettkowski, P. Wehner, E. Cutiscev, and D. Goehringer, "LinROS: A linux-based runtime system for reconfigurable MPSoCs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, vol. 15, no. 5, May 2016, pp. 208–216.
[12] Y. Wang, X. Zhou, L. Wang, J. Yan, W. Luk, C. Peng, and J. Tong, "SPREAD: A streaming-based partially reconfigurable architecture and programming model," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 21, no. 12, pp. 2179–2192, Dec. 2013.
[13] D. Göehringer, M. Huebner, E. Zeutebouo, and J. Becker, "CAP-OS: Operating system for runtime scheduling, task mapping and resource management on reconfigurable multiprocessor architectures," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops*, Apr. 2010, pp. 1–8.
[14] X. Iturbe, K. Benkrid, C. Hong, A. Ebrahim, R. Torrego, I. Martinez, T. Arslan, and J. Perez, "R3TOS: A novel reliable reconfigurable real-time operating system for highly adaptive, efficient, and dependable computing on FPGAs," *IEEE Trans. Comput.*, vol. 62, no. 1, pp. 1542–1556, Aug. 2013.
[15] T. Xia, J.-C. Prevotet, and F. Nouvel, "Mini-NOVA: A lightweight ARM-based virtualization microkernel supporting dynamic partial reconfiguration," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops*, May 2015, pp. 71–80.
[16] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass, "hthreads: A hardware/software co-designed multithreaded RTOS kernel," in *Proc. IEEE Conf. Emerg. Technol. Factory Autom.*, Sep. 2005, pp. 71–80.
[17] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, and M. Haselman, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit. (ISCA)*, Oct. 2014, pp. 13–24.
[18] Y. Sungjoo and A. A. Jerraya, "Introduction to hardware abstraction layers for SoC," in *Proc. Design, Automat. Test Eur. Conf. Exhib.*, Mar. 2003, pp. 336–337.
[19] *Hardware Acceleration in SoC FPGAs*, Altera, San Jose, CA, USA, 2015.
[20] (May 3, 2017). *Linux Programmer's Manual—Pthreads*. [Online]. Available: http://man7.org/linux/man-pages/man7/pthreads.7.html
[21] (Feb. 2015). *Linux Programmer's Manual—Devmem*. [Online]. Available: http://man7.org/linux/man-pages/man4/mem.4.html
[22] (Mar. 2017). *Linux Programmer's Manual—Mmap*. [Online]. Available: http://man7.org/linux/man-pages/man2/mmap.2.html
[23] A. Guerrieri, S. Kashani-Akhavan, B. Belhadj, P. Lombardi, and P. Ienne, "A dynamically reconfigurable platform for high-performance and low-power on-board processing," in *Proc. NASA/ESA Conf. Adapt. Hardw. Syst. (AHS)*, 2018, pp. 74–81.
[24] *Vivado High-Level Synthesis*. Accessed: 2018. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html
[25] *Intel High Level Synthesis Compiler*. Accessed: 2018. [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html
[26] R. Richter, J. Louis, and B. Berthelot, *Sentinel-2 MSI—Level 2A Products Algorithm Theoretical Basis Document*, document S2PAD-ATBD-0001, 2011.

ANDREA GUERRIERI received the M.Sc. degree in electronic engineering from Politecnico di Torino, Italy, in 2015. Since 2006, he started working on embedded systems, in PB Elettronica, Italy, where he became a Principal Engineer responsible for the development of the company's flagship products. In 2017, he joined the Processor Architecture Laboratory at École Polytechnique Fédérale de Lausanne, where he leads research projects in collaboration with industry. Recent projects involve reconfigurable SoCs exploiting dynamic partial reconfiguration of FPGAs for future space missions and planet observation.

SAHAND KASHANI-AKHAVAN received the B.Sc. and M.Sc. degrees in computer science from École Polytechnique Fédérale de Lausanne, in 2014 and 2017, respectively. He is currently pursuing the Ph.D. degree with the Very Large Scale Computing Laboratory (VLSC), EPFL under the supervision of Prof. J. Larus and works on hardware-friendly algorithms for scale-out bioinformatics workloads.

MIKHAIL ASIATICI received the B.Sc. degree in electronic engineering from Politecnico di Torino, in 2012, and the M.Sc. degree in nanotechnologies for ICTs from Politecnico di Torino, INP Grenoble and EPFL, in 2014. From November 2014 to August 2015, he was a Ph.D. student with the KTH Royal Institute of Technology in Stockholm, Sweden, where he worked on inertial MEMS sensors and integration for high temperature applications. He is currently pursuing the Ph.D. degree with the Processor Architecture Laboratory, École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland. His research interest includes memory systems for parallel accelerators with irregular memory access patterns.

PAOLO IENNE received the Ph.D. degree in computer science from the École Polytechnique Fédérale de Lausanne. He is currently a Professor with the School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne, where he heads the Processor Architecture Laboratory. His research interests include computer and processor architecture, electronic design automation, computer arithmetic, FPGAs and reconfigurable computing, and multiprocessor systems-on-chip. He is an Associate Editor of the ACMC SUR and the ACM TACO.