Split Boot—True Network-Based Booting on Heterogeneous MPSoCs

Marvin Fuchs, Luis E. Ardila-Perez, Torben Mehner, and Oliver Sander

***Abstract*— In the context of the high-luminosity (HL) upgrade of the large hadron collider (LHC), many custom ATCA electronic boards are being designed containing heterogeneous system-on-chip (SoC) devices, more specifically the Xilinx Zynq UltraScale+ (ZUS+) family. While the application varies greatly, these devices are regularly used for performing board management tasks, making them a fundamental element in the correct operation of the board. A large number of hundreds of SoC devices create significant challenges in their firmware deployment, maintenance, and accessibility. Even though U-Boot on ZUS+ devices supports network boot through the preboot execution environment (PXE), the standard ZUS+ boot process contains application-specific information at earlier boot steps, particularly within the first-stage bootloader (FSBL). This prevents the initialization of several devices from a universal image. Inspired by the PXE boot process on desktop PCs, this article describes split boot, a novel boot method tailored to the specific needs of the ZUS+. All application-specific configuration is moved to a network storage device and automatically fetched during the boot process. We considered the entire process, from firmware and software development to binary distribution in a large-scale system. The developed method nicely integrates with the standard Xilinx development toolset workflow.**

***Index Terms*— Booting, large-scale experiments, multiprocessor system-on-chip (MPSoC), network booting, preboot execution environment (PXE), system-on-chip (SoC), Zynq ultrascale+ (ZUS+).**

I. Introduction

THE Xilinx Zynq UltraScale+ (ZUS+) devices are heterogeneous multiprocessor system-on-chips (MPSoCs) that, in addition to the programmable logic (PL), contain a processing system (PS) with a number of hard processing units, such as an ARM Cortex-A53 named application processing unit (APU), an ARM Cortex-R5 named real-time processing unit (RPU), and the platform management unit (PMU) based on the MicroBlaze architecture [1]. Even though not all processors have to be involved in the boot process, it usually relies on several of them. To make the ZUS+ devices deployable in a wide range of applications, they are designed to be highly configurable. To a certain extent, this also applies to the boot process, as shown in Fig. 1. For example, it is possible to load both, the bitfile for the PL and the firmware for the RPU, in either the first stage bootloader (FSBL), the second-stage boot loader U-Boot or from Linux. In some cases, it is also possible to change the order, for example, to load the PMU firmware either before or after the FSBL.

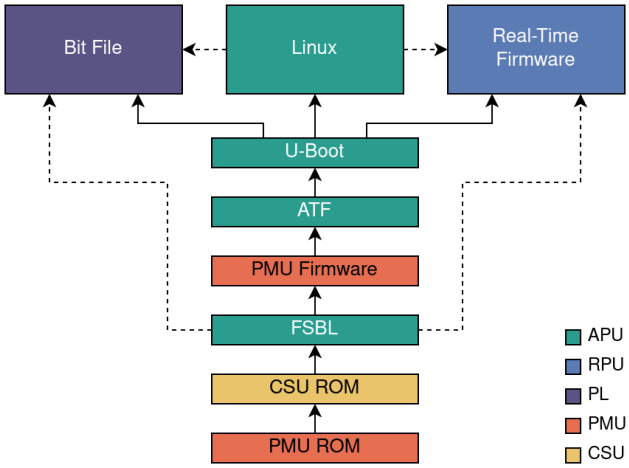

Fig. 1. Default software stack used to boot Xilinx Zynq UltraScale+ devices. The solid lines describe one possible example of a boot process, whereas the dashed lines show alternative possibilities to load a bitfile for the PL and the firmware for the RPU.

The first stage in the boot process that contains application-specific information is the FSBL. Vivado generates a C code to configure the PS and to optionally divide it into multiple subsystems via an isolation configuration according to the settings selected in the Vivado PS configuration wizard (PCW) GUI [2]. When the FSBL software project is set up, this source code is automatically integrated.

The FSBL contains information on how to configure the various internal clocks, the interfaces to the PL, and external interfaces such as the universal asynchronous receiver-transmitters (UARTs) or the network. The application-specific code contained in it to do this is one of the reasons why the FSBL cannot be factory-saved to a nonvolatile memory within the MPSoC. Thus, it is one of the first components loaded from external storage (e.g., a quad serial peripheral interface (QSPI) memory or an SD Card). The same storage location is usually also used for the PMU firmware, the arm trusted firmware (ATF), and U-Boot. In contrast, entirely generic software like the PMU ROM and the configuration security unit (CSU) ROM is stored on nonvolatile memory within the ZUS+ [3].

This work was supported by the Doctoral School "Karlsruhe School of Elementary and Astroparticle Physics: Science and Technology."

The authors are with the Institute for Data Processing and Electronics (IPE), Karlsruhe Institute of Technology, 76344 Eggenstein-Leopoldshafen, Germany (e-mail: marvin.fuchs@kit.edu).

The goal of the modified boot process presented in this article is to fetch all application-specific data including the PS configuration from a network source. This is, however, not trivial because the FSBL itself is a low-level stage in the boot process and it is not designated to communicate via a network connection. Just as with PCs that boot via preboot execution environment (PXE), the greatest advantage for MPSoCs that obtain all application-specific data from the network appears in large systems where many devices need to be maintained. One example is the distribution of updates in a large and distributed system, comprised of many identically configured boards. In its most efficient implementation, split boot enables accomplishing the task by only updating the single network storage and rebooting the devices. Significant time is saved compared to having to flash the local storage of each board. In the remainder of this contribution, we present a two-step approach that essentially supports fetching the entire PS configuration via network and applying it to the PS.

II. Related Work

U-Boot already features PXE, which allows devices to boot into an operating system such as Linux via the network. However, it does not cover the configuration of the PS [4]. Such functionality is not provided, because PXE was originally designed for computer networks rather than networks of highly configurable system-on-chips (SoCs) such as the ZUS+ family. Modifications to the FSBL on MPSoC devices might be required regularly due to containing application-specific information. This is a major drawback compared to the boot of a desktop PC, where updates to the basic input–output system (BIOS) and all other software used before the second-stage boot loader are very rarely necessary. Research regarding PXE usually targets desktop PCs [5] or servers [6], but not embedded devices. Xilinx provides means to adapt the boot process based on the application domain [1] and further describes in a patent the boot process possibilities of MPSoC devices [7]. However, the ability to load the PS configuration from a remote location is not mentioned. In the context of the high-luminosity (HL) upgrade of the large hadron collider (LHC), active research is being conducted about the boot process of MPSoC devices. To date, though, work has focused primarily on investigating and securing the possibilities provided by Xilinx [8] and building the Linux distribution for use on the device [9].

III. Split Configuration Approach

Simply moving the entire configuration of the PS part of a ZUS+ MPSoC to network storage is not feasible. Some initial configuration is needed to bring up the essential functionality of the device. This includes, first and foremost, the network interface and the configuration of the DDR memory controller, but also some internal configuration. While this configuration is board-specific, it is not application-specific. In conclusion, we propose using a base configuration that is static and reduced to the absolute minimum to boot into U-Boot with network access. This approach is similar in many ways to that

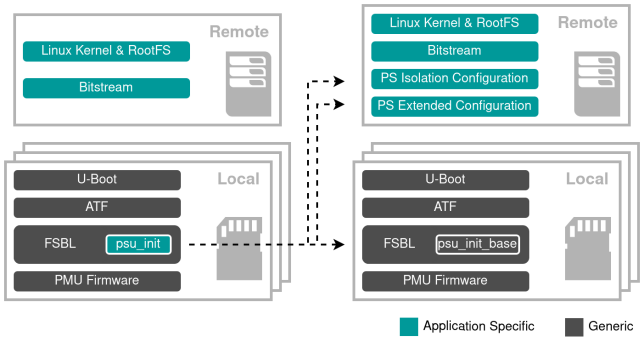

Fig. 2. Split boot approach removes from the FSBL the application-specific PS configuration and the isolation configuration. As a result, only generic information remains on the local boot medium.

of a PC BIOS. Updates to the base firmware are possible, but they are expected to happen rarely.

The application-specific data for the PS can be split into two tasks. The first one includes the configuration of all the individual components like clocks within the PS, the PS-PL interfaces, and some peripherals like the DDR memory. The second task is the application of the so-called isolation configuration, which divides the PS into multiple subsystems and defines access permissions between them. For both, we propose to move the application-specific data into separate binary configuration files, which are then fetched from a network source and applied by U-Boot during the boot process. Fig. 2 shows how removing the application-specific configuration data from the FSBL leads to a system where only generic software remains on the local boot medium and everything application-specific is stored on the network.

IV. Modified Boot Process

Xilinx ZUS+ devices require multiple tweaks in the default boot process to allow for changes to the PS configuration after it has been initially configured in the FSBL. An overview of these modifications is provided by the example boot sequence in Fig. 3, where the changes are shown as white boxes with red frames. The boot procedure starts as usual with the software components PMU ROM and CSU ROM stored on nonvolatile memory within the ZUS+. Afterward, the PMU firmware is started, in this case, before the FSBL.

The FSBL contains the first modification in the proposed boot process. Usually, at this stage, the PS gets initialized with its complete configuration. During the split boot process, however, this is where the PS receives its base configuration (psu_init_base), which includes the boot-related peripheral and memory configuration.

The source code that is used to configure the PS is generated by Vivado according to the options selected in the PCW and inserted into the FSBL automatically. Because of this, the software architecture of the FSBL allows us to easily replace this part of the source code, for example, with that of our generic base configuration.

When the FSBL has finished its execution, it hands over to the ATF. The ATF is a reference implementation of an ARM secure world software provided by Xilinx, and in the example

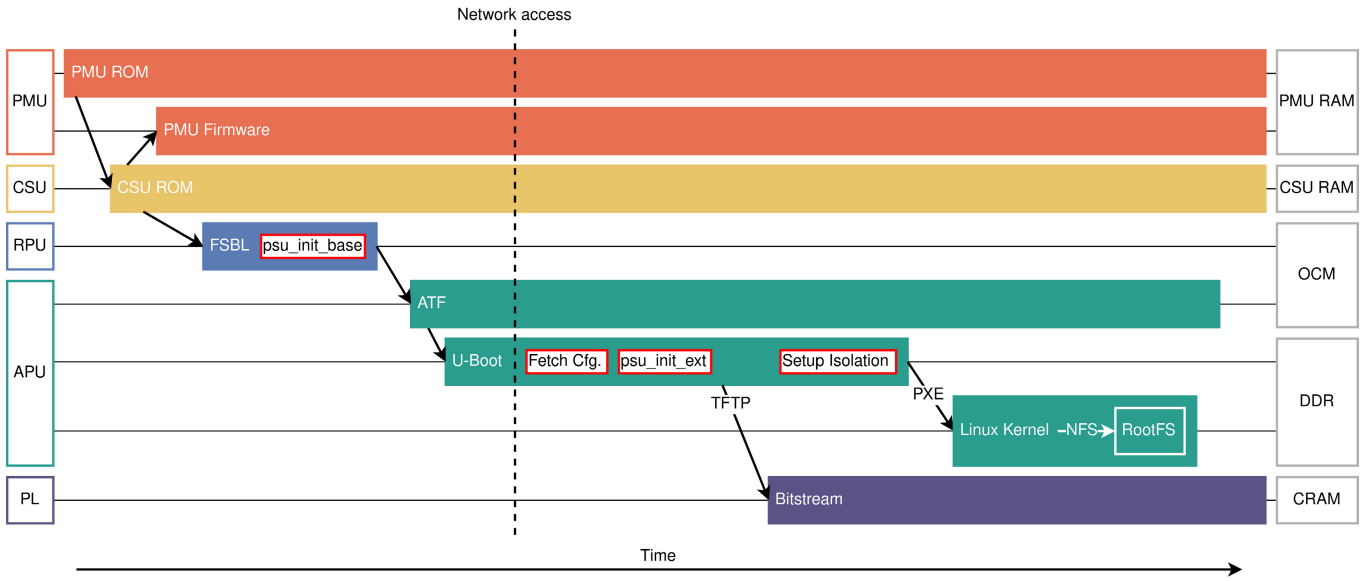

Fig. 3. Modified boot process for Xilinx ZUS+ devices. The modifications are represented by the red-bordered boxes. The first modification in the boot process, *psu_init_base*, marks were usually the configuration of the PS would happen. Here, the source code to completely configure the PS is replaced by the source code that applies a base configuration, allowing to continue the boot process and using a network interface later on. The remaining modifications can all be found in U-Boot. They include fetching configuration data, modifying the configuration of the PS (*psu_init_ext*), and optionally configuring the isolation within the PS.

depicted in Fig. 3, the first software executed on the APU. Such software is necessary to utilize the Armv8-A Exception Model that is implemented in the APU [10].

When the ATF is running, U-Boot can be started. It contains the remaining modifications to the original boot process. The first modification (Fetch Cfg.) loads all the required configuration data using standard U-Boot features from a trivial file transfer protocol (TFTP) server. This includes, in addition to the regular Linux kernel, a custom binary file for the modification of the PS configuration, a file to configure the isolation within the PS, and the PL bitstream. The next step is to extend the configuration of the PS to its complete state (psu_init_ext). U-Boot running on ARM Exception Level 2 is not allowed to access the required configuration registers directly. Instead, it is possible to request the ATF running at Exception Level 3 via a secure monitor call (SMC) to instruct the PMU firmware to access a configuration register. The PMU has unrestricted access to all configuration registers within the PS [3]. Using this methodology, the ATF can remain unmodified, while only a minor modification to the PMU firmware is required to temporarily allow U-Boot to make all these requests until the PS is fully configured.

After the reconfiguration of the PS is finished, it might be necessary to rebind some U-Boot drivers, for example, the one used for Ethernet. Then, the bitstream can be written to the PL. It is also possible to configure isolation within the PS (Setup isolation) if desired. At this point, the system is fully configured, operational, and behaves exactly the same as it would with the traditional boot. Finally, U-Boot can continue to boot Linux on the APU. In the example boot process shown in Fig. 3, PXE is used to load the Linux kernel, and the kernel itself uses network file system (NFS) to mount the root file system (rootfs).

TABLE I

LIST OF REGISTERS THAT CANNOT BE MODIFIED FROM U-BOOT [11]

Register	Mask	Reason
0xFD1A0030 0xFD1A002C 0xFD1A002C 0xFD1A002C 0xFD1A0044 0xFD1A004C 0xFD1A0080	0xFE7FEDEF 0x00717F00 0x00000008 0x00000001 0x00000002 0x00003F00 0x00003F07	Part of the configuration of the DDR.
0xFF260020 0xFF260000	0xFFFFFFFF 0x00000001	Can be read from the ATF.
0xFD0C00AC 0xFD0C00B0 0xFD0C00B4 0xFD0C00B8	0xFFFFFFFF 0xFFFFFFFF 0xFFFFFFFF 0xFFFFFFFF	SATA Port Phy configuration registers initialized with reset values [12]. Modifying these registers causes a crash.

Modifying the configuration of some critical components within the PS is not possible from U-Boot. This applies to the DDR interface and a very limited number of other configuration registers as shown in Table I. However, for many resources that are used by U-Boot but are not essential for it to run, it is possible to overwrite the configuration. Activating the isolation within the PS from U-Boot brings some limitations as well. All software that are running while the isolation is being activated must observe the restrictions enforced by them. If the isolation includes a restriction of the memory ranges used by the APU, for instance, it is mandatory that U-Boot observes this restriction before the activation of the isolation. Otherwise, U-Boot might lose access to essential data when the isolation is activated, which can lead to undesired behavior or even to a crash of the system.

TABLE II
LIST OF FUNCTIONS REPRESENTED IN THE BINARY CONFIGURATION FILES

Name	Action
PSU_Mask_Write	Read-modify-write
mask_poll	Polls until a 1 occurs in the masked part of the register or a specified number of attempts in exceeded
mask_pollOnValue	Polls until the masked part of the register matches the desired value, or until a certain number of attempts is exceeded
mask_delay	Delay for a specified duration
serdes_illcalib	Calibration algorithm for SerDes
serdes_fixcal_code	Calibration algorithm
serdes_enb_coarse_saturation	Activates the coarse saturation logic for PLLs of all four GT lanes of the MPSoC
psu_init_xppu_aper_ram	Initialization of the PPU

```
PSU_Mask_Write(CRL_APB_RPLL_CFG_OFFSET,
               0xFE7FEDEFU, 0x7E4B0C82U);
PSU_Mask_Write(CRL_APB_RPLL_CTRL_OFFSET,
               0x00717F00U, 0x00015400U);
PSU_Mask_Write(CRL_APB_RPLL_CTRL_OFFSET,
               0x00000008U, 0x00000008U);
PSU_Mask_Write(CRL_APB_RPLL_CTRL_OFFSET,
               0x00000001U, 0x00000001U);
PSU_Mask_Write(CRL_APB_RPLL_CTRL_OFFSET,
               0x00000001U, 0x00000000U);
mask_poll(CRL_APB_PLL_STATUS_OFFSET,
          0x00000002U);
```

Listing 1. Source code snippet from psu_init.c.

V. CUSTOM CONFIGURATION FILES

The PS configuration files that are stored and later fetched from the network are encoded using a binary format. This is done to efficiently process them in U-Boot. Despite the many features of U-Boot, it is still low-level software. Therefore, working with more complex data formats based on American standard code for information interchange (ASCII) like extensible markup language (XML) would significantly increase the overhead. While human-readable code would be an advantage, the wish to manually change about one thousand 32-bit registers is rather unlikely.

The internal structure of the binary configuration files is strongly inspired by the architecture of the source code that is used in the FSBL to configure the PS. This code can be unwrapped to a long list of calls of eight different functions listed in Table II [11]. The configuration of the PS is, therefore, fully represented by this list of function calls including the respective call arguments. Therefore, this is the only information that must be stored in the binary configuration files. Listings 1 and 2 show the encoding of the function calls in the binary format. All call arguments of the functions in Table II are 32-bit values, which can also be realized by macros in the source code. It is possible to represent each of the eight different functions with a unique 32-bit ID. As seen

```
00000001 FF5E0034 FE7FEDEF 7E4B0C82
00000001 FF5E0030 00717F00 00015400
00000001 FF5E0030 00000008 00000008
00000001 FF5E0030 00000001 00000001
00000001 FF5E0030 00000001 00000000
00000003 FF5E0040 00000002 0000000F
```

Listing 2. Encoding of the source code in Listing 1 in a binary configuration file.

in Listings 1 and 2, the function `PSU_Mask_Write` has the ID `0x00000001`. The binary file is now composed of the list of function calls from the FSBL encoded in this format. It is possible to navigate through the different function calls in such a binary file because the number of arguments of each function is constant and known. Finally, a distinct unique ID `0x0000000F` is used to mark the end of the file, as can be seen at the end of the file in Listing 2.

VI. PSU CONFIGURATION GENERATOR

A Python tool called the PSU Configuration Generator was developed to keep the effort of developing a project using the split boot mechanism to a minimum. This tool handles, among other things, the generation of binary configuration files. It was designed to integrate seamlessly with the development tools provided by Xilinx. Thus, the *.xsa [Xilinx support archive (XSA)] files exported from Vivado are used as input data. Within this archive, *psu_init.c* and *psu_init.h* contain the C source code that is used in the FSBL to configure the PS. They also contain a more abstract description of the configuration of the PS and the PL in the XML file *zusys.hwh*. The current implementation of the PSU Configuration Generator uses the *.xsa file containing the complete PS configuration as input.

The files *psu_init.c* and *psu_init.h* are parsed to identify the function calls that need to be written to the configuration files. The parser uses a depth-first search to find all calls of the eight functions listed in Table II, starting with the functions that can be called directly from the FSBL. If a function call is allowed in U-Boot, or, in other words, if the addressed registers can be modified from U-Boot, the function call is encoded in the binary format, resolving all macros contained in it, and appended to the binary output file. Furthermore, to be adaptable, the PSU Configuration Generator allows skipping selected registers or ignoring some function calls specified in a JSON file. There are only a few interfaces to the FSBL that represent root nodes. Using the functions in Table II, all nodes representing termination conditions for the depth-first search are identified. Both, the root and termination nodes, form the boundary conditions for the search algorithm.

The file *psu_init.c* contains two main interfaces to the FSBL: one to configure the PS and one to set up the isolation. To execute these two actions separately from U-Boot, the PSU Configuration Generator offers the possibility to export separate configuration files. Additionally, it provides the option to extract the bitfile for the PL from the *.xsa archive. Therefore, it achieves a higher degree of automation as all these files need to be copied to the same TFTP server.

VII. DEVELOPMENT WORKFLOW

To make split boot usable in real-world applications, it is important to integrate the required modifications to the

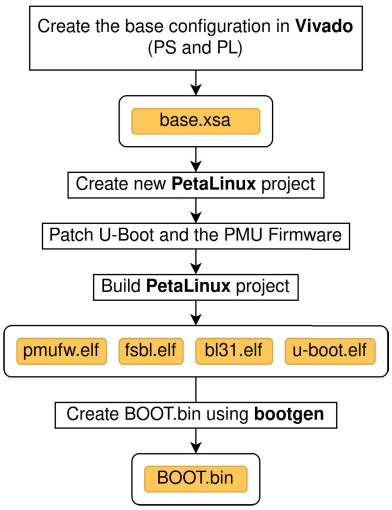

Fig. 4. Creation flow of all software and firmware components to be stored on the local boot medium. They are based on a Vivado project representing the base configuration of the PS and optionally also the PL. Output files are packed in the single boot image called *BOOT.bin*.

different software components and the additional steps in the development process with the default workflow of the Xilinx development tools. To achieve this, an approach based on two Vivado projects was chosen. One project represents the base configuration of the PS, and optionally also of the PL, which are applied at the FSBL. The other project contains the complete configuration, which is fetched by U-Boot via the network. Both Vivado projects are integrated into a workflow that is divided into two independent subprocesses. The first one leads to the generation of all files that are needed in the boot routine before network access is possible. The second subprocess produces the necessary files that can be loaded from the network. This distinct separation enables only the second subprocess to be required for each new project. The first subprocess, containing only generic data, is only executed once per hardware platform. As a result, the development effort with the split boot is comparable to a project without it.

A. Creation of the Base Configuration

The creation of all files needed for the early stages of the boot process, before a network connection can be utilized, are depicted in Fig. 4. In particular, they also contain the base configuration for the PS. As can be seen, these files are entirely based on the *.xsa file exported from the Vivado project representing the base configuration *base.xsa*. The PetaLinux tools are the only additional tools from the Xilinx development suite that are required in the process. After creating a respective PetaLinux project, the tools automate the process of building all individual software components. However, one manual step might be required if the complete configuration includes an isolation setting because the memory regions used by U-Boot need to be restricted according to it. This can be achieved in the device tree, and it is the only limitation we have thus far observed as a result of the activation of the isolation within the PS from U-Boot. As can be seen in Fig. 4, patches are used to apply the required modifications to U-Boot and the PMU firmware. The use of patch files is an integral part of developing the PetaLinux software suite, and thus both the creation and the application during the build process are automated.

The patch applied to the U-Boot source code is used to add the functionality to modify the configuration of the PS and to apply the isolation. This functionality is packaged in the custom U-Boot-command *psuinite*. As an argument, this command needs the address of the configuration file to be applied in memory. It then iterates over the configuration file and executes the function calls listed there. The command contains implementations of all functions listed in Table II. The source code is derived from the implementations in *psu_init.c* and only slightly modified to use the drivers available in U-Boot and to request access to the required configuration registers via SMC from the PMU firmware instead of accessing them directly. A flag can be passed to the command if the access to the configuration registers from the APU should be locked in the PMU firmware after the configuration file is applied. Finally, a debugging flag exists that enables print outputs for each register access made. Another patch applied to the U-Boot source code inserts all the additional steps required by a split boot to the regular steps U-Boot performs to boot the system. This patch also includes checks if the additional steps were executed successfully or not. If a failure is detected, the boot process is immediately aborted with an error message because the errorless execution of each of these steps is essential for a successful boot.

The patch applied to the PMU firmware is required to allow U-Boot to access all needed configuration registers via SMC calls. By default, the PMU firmware verifies that the requesting instance is authorized to access the requested resource. This mechanism must be temporarily disabled until U-Boot has completed all required configuration register accesses. The patch enables all such accesses from the start of the PMU firmware and gives U-Boot the option to restore the default access control when the configuration has been extended to its complete state.

PetaLinux tools are able to build the PMU firmware, the FSBL, the ATF, and U-Boot once the patches have been applied. Since this FSBL is based on the *base.xsa*, it already contains the desired configuration for the PS in *psu_init.c* and *psu_init.h*, so modifying these files is not necessary. However, the FSBL contains one more application-specific section. The structure *XPm_ConfigObject* contains, among other things, the information about which components in the PS will be used in the given configuration. One of the final steps in the FSBL is to send this information to the PMU firmware. If a component is not marked as active in this structure, it is not possible to activate it later purely via configuration registers. One workaround for this limitation is to mark every node in the *XPm_ConfigObject* as active. The downside is that this increases the power consumption of the MPSoC as all nodes will be powered, which also leads to potential security vulnerabilities. Therefore, it is still being investigated if this structure can be modified at a later stage of the boot process.

Having the PMU firmware, the FSBL, the ATF, and U-Boot ready, the final step is to use the Xilinx tool *bootgen* to package

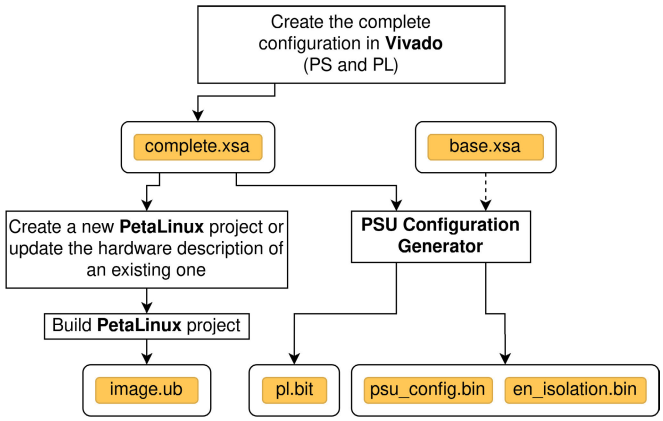

Fig. 5. Creation of all software and firmware components that are fetched from the network during boot. They are based on a Vivado project representing the complete configuration of the PS and PL. The tool *PSU Configuration Generator* was developed to automate the creation of the binary configuration files, it could optionally also use the file *base.xsa*.

them in a boot image. This file can then be copied to a local boot medium such as an SD Card.

B. Creation of the Complete Configuration

Fig. 5 shows the process to create all files used for the later stages of the boot process that can be fetched from a TFTP server in the network, including the binary configuration files to extend the configuration of the PS. It can be seen that two paths are used to create the files. One uses PetaLinux tools to build the Linux kernel and the other path uses the custom tool PSU Configuration Generator to generate the two custom binary configuration files and to extract the bitfile for the PL. In contrast to the files created for the early stages of the boot process, the files created here contain application-specific information and are thus mainly based on the *.xsa file exported from the Vivado project representing the complete configuration *complete.xsa*. The information in *base.xsa* can be used optionally to achieve a higher degree of automation.

The path in Fig. 5 for building the Linux kernel uses solely the file *complete.xsa* as input. This archive is used as the basis to set up a PetaLinux project. Afterward, the PetaLinux framework fully automates the process of building the kernel. The tools provided by the PetaLinux software suite can be used to customize the kernel as usual.

To prevent redundant reconfigurations, the path in Fig. 5 showing the usage of the PSU Configuration Generator could use both *complete.xsa* and *base.xsa*. However, currently, only the complete configuration is used as input. The redundant reconfigurations that occur because of this have not caused any problems so far. However, the registers that cannot be modified from U-Boot (see Table I) must be declared in the JSON configuration files. Fig. 5 also makes clear why it is efficient to use the PSU Configuration Generator to extract the bitfile for the PL from *complete.xsa*. This feature helps to have as many of the files that must be copied to the remote server ready at the same time and the same location. Only the Linux kernel needs to be collected from a different location.

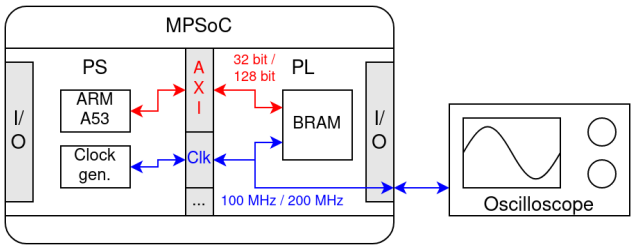

Fig. 6. Setup used to test the reconfigurability of the AXI and clock interfaces between PS and PL by the split boot mechanism. After the initial configuration, both interfaces were enabled with 32-bit AXI width and 100-MHz clock. Later, they were changed to 128-bit and 200 MHz.

VIII. Implementation and Testing

The split boot mechanism as described here was developed and tested on a Trenz Electronic TE0803-03-4BE11-A MPSoC System-on-Module (SoM) plugged onto a custom carrier board [13] that included, among other things, an solid-state drive (SSD), two UART interfaces, and two network interfaces, one via serial gigabit media-independent interface (SGMII) and one via reduced gigabit media-independent interface (RGMII). On the software side, the development tools of the Xilinx toolset 2020.2 were used.

Because the split boot process in its most efficient implementation loads the configuration for the PL from a network server, the ability to configure the interfaces between PS and PL at run time is of great interest. Two independent tests were run for validation. With the clocks generated in the PS directly connected to multiplexed input–output (MIO) pins of the PL, the ability to activate the signal and change the frequency was confirmed with an externally connected oscilloscope. The second test targets the Advanced eXtensible Interface (AXI) interfaces. A block RAM (BRAM) intellectual property (IP) core instantiated in the PL was used to confirm the possibility to activate them at run time and to change the width of the bus. Fig. 6 shows the setup used for both tests.

The reconfigurability of interfaces using serializer/deserializer (SerDes) was examined using the connected SSD. SerDes interfaces are highlighted in particular here, because they are not only configured but also calibrated by the FSBL and this calibration step was also relocated to U-Boot. After changing the configuration and performing the calibrating in U-Boot, read and write access to the SSD from Linux was possible without any limitations. Another interface using SerDes is Ethernet via SGMII. The Ethernet interface, however, needs to be configured in the FSBL because it is used in the split boot mechanism. Thus, the only test possible was to use U-Boot to clear the respective configuration registers with zeros before restoring the configuration values. This test was also successful. After rebinding the Ethernet driver in U-Boot, the interface could be used normally. The same procedure was also successfully tested with the Ethernet interface based on RGMII that consequently does not use SerDes. In addition, it was also tested whether the configuration of the MIO pins of the PS can be changed.

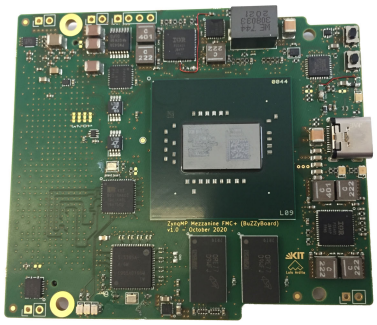

Fig. 7. Custom Zynq Ultrascale+ MPSoC-based FMC+ mezzanine board designed for slow control tasks.

For this purpose, two MIO pins were assigned to one of the UARTs in the PS at run time. After that, the UART could be used without restrictions for input and output.

Aside from these tests aiming at the configurability of a single component, booting Linux on the MPSoC after extending the configuration in U-Boot was used as a comprehensive test. This is possible because the majority of components in the PS that are configured as part of the complete configuration are targeted and initialized by a Linux driver loaded during the kernel's boot process. Linux was able to boot on the reconfigured MPSoC in the same way as if the PS had been fully configured in the FSBL. This supports the claim that after reconfiguration in U-Boot, the MPSoC behaves exactly as if the configuration had been done completely in the FSBL.

To investigate whether the isolation configured in U-Boot behaves the same way as if it had been configured in the FSBL, two types of tests were run. The access to different regions in the address range of the DDR memory, separated by the isolation, was examined before and after the isolation was enabled. A similar access check was also performed for multiple registers belonging to different isolated components within the PS. In both cases, the isolation behaved the same way as if it had been activated in the FSBL. This outcome was expected because, despite the fact that the configuration of the isolation is handled in software, the actual separation of the PS into multiple subsystems is enforced directly by the hardware and thus unaffected by the order in which the software is executed [3].

In addition to the Trenz Electronic MPSoC, split boot based on version 2020.2 of the Xilinx development tools was also implemented on a Xilinx ZCU102 evaluation board and a custom ZUS+-based FPGA Mezzanine Card Plus (FMC+) mezzanine board, depicted in Fig. 7 [14]. Furthermore, it was implemented on a Xilinx Kria K26 SoM plugged onto a KV260 development platform using version 2020.2.2 of the Xilinx toolset. Despite some minor changes to the patches required due to the different versions of the toolset used for the Kria K26, the test results were identical. The implementation process on these different hardware platforms was also used to estimate the effort required to create all the projects and files needed for a new platform. Due to the two Vivado and PetaLinux projects used, the process takes longer than with the regular boot process, but the additional time required was typically well under an hour, especially when the patches for the version of the toolset used were already available.

IX. Conclusion

A large number of hundreds of SoC devices used within the LHC upgrade creates significant challenges in their firmware deployment, maintenance, and accessibility. Booting from a singular source would be beneficial and would significantly ease maintenance. This functionality is supported by the modified boot process presented in this article. The split boot process enables a clear separation by having all application-specific data on a remote server and just a generic base layer of software remaining on the local boot medium. The proposed workflow minimizes the overhead of implementing the modified boot process while relying on official Xilinx tools wherever possible. The split boot was implemented and tested on four different hardware platforms with two versions of the Xilinx development tools. Although the boot sequence is already fully functional, there is still room for improvement. A higher level of automation could be attained and will be addressed in future work.

References

[1] *Zynq UltraScale+ MPSoC Software Developer Guide*, version 2020.2, Xilinx. https://docs.xilinx.com/r/2020.2-English/ug1137-zynq-ultrascale-mpsoc-swdev. Accessed on: August 2, 2022.

[2] *Vivado PS Configuration Wizard Overview*. Accessed: Aug. 23, 2022. [Online]. Available: https://www.xilinx.com/video/hardware/vivado-ps-configuration-wizard-overview.html

[3] Xilinx. *Zynq UltraScale+ Device Technical Reference Manual*, Version 2.3. Accessed: Nov. 11, 2022. [Online]. Available: https://docs.xilinx.com/r/en-U.S./ug1085-zynq-ultrascale-trm/Zynq-UltraScale-Device-Technical-Reference-Manual

[4] Xilinx. *PetaLinux Tools Documentation Reference Guide*. Version 2022.1. Accessed: Aug. 9, 2022. [Online]. Available: https://docs.xilinx.com/r/en-U.S./ug1144-petalinux-tools-reference-guide

[5] T. Cruz, P. Simoes, F. Bastos, and E. Monteiro, "Integration of PXE-based desktop solutions into broadband access networks," in *Proc. Int. Conf. Netw. Service Manage.*, Niagara Falls, ON, Canada, 2010, pp. 182–189, doi: 10.1109/CNSM.2010.5691309.

[6] L. Guojie and Z. Jianbiao, "A TPCM-based trusted PXE boot method for servers," in *Proc. IEEE 5th Int. Conf. Signal Image Process. (ICSIP)*, Nanjing, China, Oct. 2020, pp. 996–1000, doi: 10.1109/ICSIP49896.2020.9339366.

[7] B. A. S. Krishna, M. J. Sarmah, and A. A. V. Kumar, "Multistage boot image loading and configuration of programmable logic devices," W.O. Patent 2017/06 247 9A1, Apr. 13, 2017.

[8] N. Dzemaili, "A reliable booting system for Zynq Ultrascale+ MPSoC devices," B.S. thesis, HU Univ. Appl. Sci. Utrecht, Utrecht, Netherlands, 2021.

[9] K. S. Mor, "An embedded Linux distribution for the data acquisition hardware of the compact muon solenoid experiment at CERN," M.S. thesis, Eindhoven Univ. Technol., Eindhoven, The Netherlands, 2020.

[10] *ARM Privilege and Exception Levels*. Accessed: Aug. 2, 2022. [Online]. Available: https://developer.arm.com/documentation/102412/0102/Privilege-and-Exception-levels

[11] M. Fuchs, "Highly integrated slow control on heterogeneous SoC architectures," M.S. thesis, Karlsruhe Inst. Technol., Karlsruhe, Germany, 2021.

[12] *Zynq Ultrascale+ Devices Register Reference*. Accessed: Aug. 16, 2022. [Online]. Available: https://www.xilinx.com/htmldocs/registers/ug1087/ug1087-zynq-ultrascale-registers.html

[13] L. Ardila-Perez et al., "A novel centralized slow control and board management solution for ATCA blades based on the Zynq Ultrascale+ system-on-chip," in *Proc. Int. Conf. Comput. High Energy Nucl. Phys. (CHEP)*, vol. 245, Nov. 2020, Art. no. 01015, doi: 10.1051/epjconf/202024501015.

[14] T. Mehner et al., "ZynqMP-based board-management mezzanines for Serenity ATCA-blades," *J. Instrum.*, vol. 17, no. 3, Mar. 2022, Art. no. C03009, doi: 10.1088/1748-0221/17/03/C03009.