Evaluation of a Declarative Linux Kernel FPGA Manager for Dynamic Partial Reconfiguration

Ulrich Langenbach
Beuth University of Applied Sciences Berlin

Stefan Wiehler
University of Ulm

Endric Schubert
University of Ulm

***Abstract*—Heterogeneous Multi-Processor Systems-on-Chip, whether ARM or x86 based, promise further performance scalability by complementing temporal compute in CPUs/GPUs with spatial compute in digital circuitry. Dynamic partial reconfiguration (DPR) extends such compute architectures by making use of different spatial compute elements over time. Novel research [1] presents means for operating DPR by the Linux kernel via so-called device tree overlays (DTOs) and, thereby, opens up FPGA-based acceleration for general purpose computing (GPC). Operating systems (OSs) for General Purpose Computing need to administer these reconfigurable FPGA resources and, thus, need metrics for handling the trade-offs between performance gains by acceleration, FPGA resource requirements, and turn-around-times within DPR context switches. Unfortunately, precise parameters for these metrics cannot be derived by theoretical analyses but only by experimental results. Based on exemplary implementations of programmable accelerators for the Linux Kernel Crypto-API, implemented on the Xilinx Zynq 7000 platform, we are able to present first OS metrics for DTO-based DPR. These metrics can guide future research on OSs for GPC on FPGAs.**

I. Introduction

The use of heterogeneous systems promises additional performance scalability. Such heterogeneous systems consist of multiple compute elements for spatial and temporal computation [2]. Reconfigurable heterogeneous systems combine central processing units (CPUs) and field programmable gate arrays (FPGAs). FPGAs often offer dynamic partial reconfiguration (DPR) to adapt the system to new requirements or user demands over time. General purpose computing (GPC) can significantly profit from DPR, because GPC focuses on providing compute platforms for a wide range of different workloads. As GPC systems run general purpose operating systems (OSs) and demand extra compute performance, these GPC OSs now also need to support reconfigurable systems resources efficiently and dependably.

This means that the overhead created by OSs, the runtime environment, as well as the infrastructure on the FPGAs to support heterogeneous systems need to be as least intrusive as possible. Because DPR support is a critical part for heterogeneous systems for both users and operators, a detailed knowledge about the trade-offs of DPR becomes necessary. These trade-offs include the performance gain by adding programmable-function accelerators (sometimes called offload engines), the resource costs for those accelerators, and the time to reconfigure the FPGA portion to implement a different accelerator.

The Linux kernel is well known and widely used, because it has proved to be a reliable and high performance base for various applications on various platforms. As such a versatile piece of software it is especially well suited for running GPC platforms. For supporting the increasing number of heterogeneous systems, the Linux kernel just recently received the Linux Kernel FPGA Framework [1]. The implemented approach supports DPR in a vendor neutral way. It also provides a declarative reconfiguration approach for its use by exposing its functionality through the device tree (DT). The aforementioned reasons make the stock Linux kernel a selected choice for heterogeneous, reconfigurable processing platforms, but also raise the demand for researching dependable OS metrics.

Previous work shows that the Linux Kernel FPGA Framework generally offers a low overhead support for reconfigurable system [3]. However a metric for the reconfiguration process is needed to tailor applications for using heterogeneous systems as well as for optimizing operating system support for GPC workloads. Such a metric shall be based on the performance gains for a GPC system when combining temporal and spatial compute elements. Based on the size of the accelerator the reconfiguration frequency can be partially estimated. However some parts of the overall process highly depend on application drivers, so that these have to be measured or estimated separately. To provide an example application of this metric we present a detailed analysis of the reconfiguration process based on measurements of a proof-of-concept system.

The remainder of this paper is organized as follows: Section II outlines the contribution of our work. Section III introduces the Linux Kernel FPGA Framework and compares it to other approaches. A novel metric for measuring the efficiency of the reconfiguration process is introduced in Section IV. Section V briefly describes the implementation. In Section VI the evaluation and measurement results of the system are shown and compared to previous work. We provide an outlook on future work in Section VII and conclude with Section VIII.

II. Contribution

We present a performance evaluation of the Linux Kernel FPGA Framework [1] and the results are compared with previous approaches. These exemplary results are based on accelerators for the Linux Crypto-API. The accelerator FPGA blocks have been derived using high level synthesis (HLS)

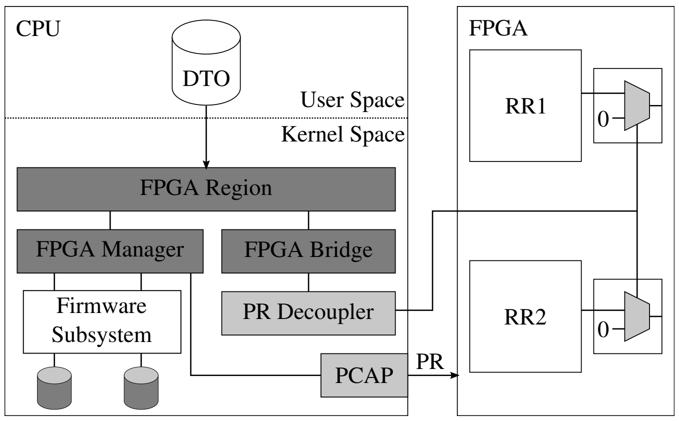

Fig. 1. Block Diagram of the Linux FPGA Framework

from software reference implementations for the corresponding Crypto-API function. As per our knowledge this is the first of such evaluations. This can be used to enhance schedulers, or governors, for reconfigurable GPC systems or as a methodology when mapping applications onto such systems.

III. Related Work

First, in section III-A we will give an overview over key research work within the very wide area of reconfigurable computing. We present the most recent and most corresponding contributions related to this work. Then, we give an introduction into the Linux Kernel FPGA Framework in Section III-B. Finally, the concept of the so-called device tree overlay (DTO) given in Section III-C.

A. The Reconfigurable Computing Ecosystem

A number of approaches present operating system support for reconfigurable computing. Some present special task models, such as [4], [5], [6], [7]. Unfortunately, the task models for co-scheduling of hardware and software tasks are not suitable for GPC systems, because the users need to adapt their programming models to fit the task model of the scheduler. Additionally a priori knowledge about applications is not available in GPC systems, which does not allow for optimal, but best-effort scheduling.

Other approaches [8], [9] are based on custom hardware resources built into the FPGA part. Infrastructure hardware resources which tightly interact with the OS cannot always be integrated into the reconfigurable part of GPC systems. Additional approaches are working towards integrating reconfigurable computing support into special OSs [10], [11]. As GPC computing must rely on widely used OSs to support a large user base, such approaches are not well suited for GPC, either. A more generic enhancement for the Linux kernel has also been presented in [12]. Its broader focus is different than that of the Linux Kernel FPGA Framework as it also adds an abstraction for the underlying interfaces, e.g. JTAG, automatic detection and identification of present devices as well. However, its implementation is not publicly available.

A larger number of publications focuses on specific aspects of reconfigurable systems. Some research work addresses accelerating the reconfiguration process itself [13], [14]. The former has also been ported to Linux [15], but did not receive much attention from industry. Another way to improve the utilization of a reconfigurable region (RR) is to prefetch a new configuration of a RR into its configuration memory before bringing it online. Several research teams have been working on employing such methods for more efficient hardware use [16], [17]. However, the implementation of these methods is not well supported by the tools currently available.

An ongoing research topic is scheduling of reconfigurable systems [18], [19], [20]. As the Linux Kernel FPGA Framework does not provide any scheduling means, but at the same time neither constrains external schedulers, this paper will not delve deeper into this topic. As software tasks are often executed on preemptively scheduled systems of preemptive reconfigurable computing has also been well researched [21], [22]. This topic stands out as, especially for GPC, optimizing the utilization may benefit from swapping applications running on reconfigurable resources. The Linux Kernel FPGA Framework is also open to such aspects, which will be the subject of future work.

Finally, FPGA device vendors provide a simple way to expose partial reconfiguration (PR) capabilities through an OS [23] interface. These are well-known in the research community and need no further elaboration.

B. Linux Kernel FPGA Framework

The recently available Linux Kernel FPGA Framework addresses key issues related to reconfigurable device operation such as vendor- and device-neutral interfaces as well as dependable operation. In Figure 1 the components of the Linux Kernel FPGA Framework (dark gray) are shown with corresponding low level drivers (light gray) and infrastructure within the FPGA (light gray) relevant for a Xilinx Multiprocessor System-on-Chip field programmable gate array (MPSoC-FPGA) based system.

A FPGA region represents a compute resource provided by a single RR. It also provides the entry point for user interaction. Therefore for every RR a corresponding device tree node is added to the base DT of the platform. A DTO overlaying such a node triggers the Linux Kernel FPGA Framework machinery to reconfigure the corresponding RR. The device specific drivers, implementing the frameworks device application programming interface (API) for both the processor configuration access port (PCAP) interface and the PR decoupler enable the Linux Kernel FPGA Frameworks operation on a specific device. The former is operated by the frameworks FPGA manager and the latter by the FPGA bridge components.

The concepts within the Linux Kernel FPGA Framework show generic support for reconfigurable devices that do not imply any task or hardware / software interaction models. However external frameworks may implement these schemes and employ the Linux Kernel FPGA Framework for low level device management.

C. Device Tree Overlays

In order to enable code reuse across multiple platforms the concept of the DT has been introduced to separate data from code [24], [25]. The DT describes the systems architecture including devices, its parameters and drivers in a textual, human readable way. It is compiled to a binary representation, which is interpreted by the Linux kernel at boot time.

Clean and vendor neutral interfaces to the user are provided by using the DTO mechanism [26]. It allows for providing a platform description from outside the kernel context, e.g. for vendors, Linux distributions or users. In the context of reconfigurable systems the concept of a DTO is used to modify the system itself via reconfiguration, rather than reflecting a change reported from outside of the system. After the reconfiguration process completed, the requested change is reflected by the updated DT. Via a DTO a requester of a reconfiguration provides all needed information to the kernel subsystems to manage the reconfiguration itself and the basic system setup, such as loading corresponding drivers.

IV. Metric

Based on past research and our ongoing experiments with DPR systems we introduce a metric for describing the time behavior of the Linux Kernel FPGA Framework, using the following steps. These steps (and their respective latencies) within the reconfiguration flow for one RR, in order of their execution, are:

LDTO (T_{LDTO})	The DTO of the region is loaded, e.g. by an application.
LFW (T_{LFW})	The bitstream specified by the DTO is loaded via the Linux firmware interface.
DCPL (T_{DCPL})	The RR is decoupled from the static infrastructure within the FPGA fabric.
LCFG (T_{LCFG})	The RR configuration, i.e. partial bitstream, is downloaded into the FPGA fabric.
CPL (T_{CPL})	The RR is coupled to the FPGA static DPR infrastructure.
UPDT (T_{UPDT})	The DTO containing the reconfigured RRs parameters is applied to the DT.
LDRV (T_{LDRV})	The driver for the new device, now present in the RR, is loaded.
EXE (T_{EXE})	The application executes, taking advantage of the accelerator in the RR.
UDTO (T_{UDTO})	The DTO of the RR is unloaded.
UDRV (T_{UDRV})	The driver is unloaded.
DCPL (T_{DCPL})	The RR is deactivated.
IDL (T_{IDL})	The system is idle.

The steps of the reconfiguration process as described above can be categorized into DT (T_{DT}), drivers (T_{DRV}), firmware (T_{FW}) and configuration (T_{CONF}), see equations 1 to 3. The overall time needed for the reconfiguration is defined as T_{TOT}. The time added by the Linux Kernel FPGA Framework (T_{LKFF}) is obtained by subtracting all categorized sublatencies from the total time measured as shown in Equation 4.

$$T_{DT} = T_{LDTO} + T_{UPDT} + T_{UDTO} \tag{1}$$
$$T_{DRV} = T_{LDRV} + T_{UDRV} \tag{2}$$
$$T_{FW} = T_{LFW} \tag{3}$$
$$\begin{aligned} T_{LKFF} = T_{TOT} - T_{DT} - T_{DRV} - T_{LFW} \\ -T_{DCPL} - T_{LCFG} - T_{CPL} \end{aligned} \tag{4}$$

The steps may additionally be summarized within phases with respect to their time-wise occurence, namely the configuration (T_{CONF}) and the deconfiguration (T_{DCONF}) phase as expressed in equations 5 and 6 respectively.

$$\begin{aligned} T_{CONF} = T_{LDTO} + T_{LFW} + T_{DCPL} + T_{LCFG} \\ +T_{CPL} + T_{UPDT} + T_{DRV} \end{aligned} \tag{5}$$
$$T_{DCONF} = T_{UDTO} + T_{UDRV} + T_{DCPL} \tag{6}$$

T_{LCFG} can be calculated using static parameters of a heterogeneous platform, namely the bandwidth of the PCAP infrastruture BW_{PCAP} and the applications partial bitstream size of the accelerator S_{FW}. This is shown in Equation 7.

$$T_{LCFG} = \frac{S_{FW}}{BW_{PCAP}} \tag{7}$$

The same holds true for T_{LFW}, which depends on the disk (BW_{DSK}) or random access memory (RAM) (BW_{RAM}) speed and may be calculated as shown by Equation 8. The required data about the bandwidth can easily be measured experimentally within a system, e.g. by using memory or disk benchmarks. Depending on the RAM size available and memory activity of the system the file is usually in the page cache after loading it initially, so that in average the cached timing can be assumed.

$$T_{LFW} = \begin{cases} \dfrac{S_{FW}}{BW_{DSK}}, \text{ uncached} \\ \dfrac{S_{FW}}{BW_{RAM}}, \text{ cached} \end{cases} \tag{8}$$

The remaining parameters T_{CPL} and T_{DCPL} of T_{LCFG} mostly depend on the driver implementation of the reconfigurable device and the access latency from the CPU to the device. Both steps generally show similar, if not identical, latencies as they are usually implemented by a single register access to the static RR infrastructure within the FPGA.

A major challenge are the latencies contributed by the drivers (T_{DRV}) which cannot be estimated very precisely but need careful analysis, for example via measurements within an example system.

Based on the above-introduced latency parameters the overall latency of an application based on a reconfigurable accelerator ($T_{TOT,DPR}$) is given in equation 9. Equation 10 defines the break even point for using partial reconfiguration, which is reached when a hardware accelerated solution needs less or the same time to complete the same work as a pure software solution. Equation 11 puts this break-even point in

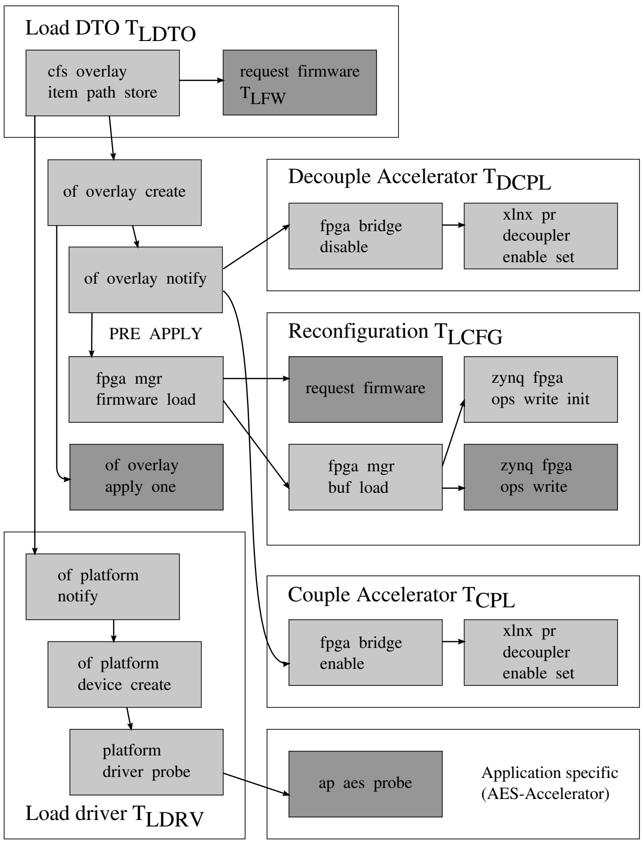

Fig. 2. Main functions of the configuration call graph

relation to the speed-up F_{SUP} gained from partial offloading the application to the FPGA.

$$T_{TOT,DPR} = T_{TOT} + T_{EXE} \tag{9}$$

$$T_{TOT,DPR} \leq T_{TOT,SW} \tag{10}$$

$$T_{TOT,DPR} \leq T_{TOT} + T_{TOT,SW} \cdot F_{SUP} \tag{11}$$

The presented metric introduces a decision base for a potential GPC scheduling algorithm or governor operating reconfigurable computing resources on heterogeneous computing platforms.

V. Implementation

To analyze the performance of the reconfiguration process we implemented a proof-of-concept system based on the commercially available *ZC706* MPSoC-FPGA platform [27]. The system is running a fully featured Xilinx Linux kernel *xilinx-v2016.1* based on vanilla Linux kernel 4.4 and an Ubuntu 15.04 based *Linaro developer for ARM* user land as provided by Linaro. The kernel has been patched with the Linux Kernel FPGA Framework patches from the corresponding mainline kernel development branch. The accelerated Linux kernel Crypto-API exposes its capabilities to kernel space services such as *dm-crypt* and user space applications as well, e.g. via *cryptodev* to *OpenSSL* [28]. The accelerator itself has been implemented using *Xilinx Vivado HLS 2016.1* [29]. The advanced encryption standard (AES) implementation [30] has been adapted to the needs of the HLS tool, while special care was taken to assure functional correctness.

Measurements of the datapath for evaluation of the accelerators performance within this setup are taken by augmenting the system with Linux kernel tracepoints. Most tracepoints were already present by default, but others have been inserted for more detailed insight. The same method is employed for the analysis of the reconfiguration process. The trace results are retrieved by *ftrace* [31]. *TRAPpy* [32], a python based ftrace result evaluation tool, is used to calculate bandwidths. The callgraph analysis is done by custom tools.

TABLE I
Overview of the latency of single configuration steps

Step	Latency [ms]	Portion [%]
T_{LFW}	27.0	20.0
T_{DCPL}	0.002	0.0
T_{LCFG}	87.0	64.4
T_{CPL}	0.001	0.0
T_{UPDT}	1.0	0.7
T_{LDRV}	18.0	13.3
Other	2.0	1.5
T_{TOT}	135.0	100.0

VI. Results

The relevant phases of an accelerators duty cycle with respect to the Linux Kernel FPGA Framework in general and specifically its overhead are the configuration `CONF` and deconfiguration `DCONF` phases. Our results are therefore shown within the next two subsections for each of the two phases, followed by a comparison with other available data.

A. Configuration

Table I summarizes the single latencies and portions in percent of the steps contributing to a total T_{TOT} of $135\,\text{ms}$ configuration time needed for a partial bitstream size of $5.9\,\text{MB}$. The latencies of all steps interacting with the FPGA are mentioned as well as the steps contributing the largest latencies. The other entry sums up latencies that are not explicitly shown in the table, because they have not been explicitly measured within our setup. When the process is executed for the first time after the system startup, step `LFW` takes longer due to the firmware being not yet loaded into the page cache.

The function call graph shown in Figure 2 visualizes how the functions work together. It is derived from the measurement traces and assembles the main contributers and functional blocks. The boxes represent the steps of the reconfiguration process. The functions within those boxes are the most important or the most time intensive ones of the respective step.

The three main contributing steps load bitstream `LFW`, reconfiguration `LCFG` and loading the driver `LDRV` are shown in more detail in the order they are executed. Because the dominating part of the reconfiguration process is the reconfiguration itself, the time needed to accomplish this task is put into relation to other solutions. Table II shows an overview

TABLE II
RECONFIGURATION TIME (T_{CONF}) FOR A 1 MB SIZED PARTIAL BITSTREAM

Solution	Bandwidth [MB/s]	Time [ms]
Theoretical PCAP capability [23, p. 21]	400	2.5
Theoretical secure mode PCAP capability [23, p. 21]	400	2.5
Typical PCAP capability [33, p. 220]	145	0.7
Typical PCAP capability [23, p. 21]	44	22.7
Theoretical ICAP capability [33]	400	2.5
ZyCap [15]	382	2.6
Our implementation	100	10.0

about how different approaches perform in comparison. Because these approaches measurements are based on different bitstream sizes, the respective results are normalized to 1 MB for comparison. It is notable that despite equal theoretical throughput provided by the `ICAP` and `PCAP` interfaces they show quite different real results. This is due to an interconnect limitation within the advanced extensible interface bus (AXI) interconnect to the `PCAP` function, specifically its assigned direct memory access (DMA) engine to the RAM [33, p. 220].

The bitstream loading step `LFW` is the same for all systems and its speed depends on the used transport technology for the bitstreams. Relevant real implementations will use some kind of non-volatile storage and transfer the bitstream to RAM during the load bitstream step `LFW` at a speed depending on the type of storage / RAM and the system interconnects.

Loading the driver in step `LDRV` differs from the other steps as it is outside of the Linux Kernel FPGA Frameworks scope. The latency of the driver depends on the number and complexity of parameters read from the DT and processed by the driver as well as the complexity of the accelerators initialisation procedure. Both subtasks are specific to the accelerator and its configuration, which means careful examination is necessary on a case to case basis. In our example system the driver initialisation latency sums up to 18 ms representing 13.3 % of the overall latency of the reconfiguration process.

B. Deconfiguration

The deconfiguration phase `DCONF` following the execution step `EXE` consists of fewer steps as compared to the configuration phase. Its latency T_{DCONF} of 0.71 ms is small and its major contributor actually is the device driver and Crypto-API clean-up. The `of_platform_device_destroy()` function within this step contributes about 0.36 ms. Depending on the driver and additionally associated frameworks, this step might need significantly more time to release the device. The reconfigurable computing framework additionally needs 0.03 ms to interact with the static infrastructure within the FPGA.

C. Discussion

The overhead of the Linux Kernel FPGA Framework ($< 2\,\%$) is small enough for a lot of use cases that reconfigure infrequently. This is also true for developers of accelerators using the Linux Kernel FPGA Framework to update their hardware during development cycles. The main contributors to the overall process latency T_{TOT} are due to loading the bitstream into RAM `LFW` ($T_{LFW} \approx 20\,\%$), the reconfiguration itself `LCFG` ($T_{LCFG} \approx 65\,\%$) and the driver parts `DRV` ($T_{LDRV} \approx 13\,\%$). The overhead with respect to the execution time of the accelerator itself is mostly determined by the reconfiguration time T_{CONF}, which is dominated by the time to write the partial bitstream into the FPGA fabric T_{LCFG}.

The overall latency T_{TOT} is the time for an accelerator request triggering the reconfiguration to become available. This needs to be taken into account for workload execution.

VII. OUTLOOK

The performance of the implemented example system and the underlying Linux Kernel FPGA Framework can be further enhanced as shown by the comparison to other proposed solutions. However the enhancement of configuration speed is specific for the underlying Zynq 7000 device. A comparison to more recent devices, such as Xilinx Zynq UltraScale+ devices or less tightly connected systems that do reconfiguration via periphal component interconnect express (PCIe), is planned.

The first step of the configuration phase (`CONF`), loading the bitstream (`LFW`), can further be enhanced for longer running systems by caching those files in RAM within the firmware cache.

Some research proposes the technique of configuration prefetching to overlap configuration time with execution time. Whether such a feature can be implemented within the reconfigurable computing framework is not yet clear. For now it seems to be a vendor dependent solution and also needs specially prepared partial bitstreams [16].

VIII. CONCLUSION

The Linux Kernel FPGA Framework provides a generic vendor neutral interface whilst maintaining low runtime overhead ($< 2\,\%$) compared to the time needed to do partial reconfiguration ($T_{LCFG} \approx 65\,\%$), or bitstream fetching from storage ($T_{LFW} \approx 20\,\%$). These significant delays in the reconfiguration, although not part of the reconfiguration framework, can be optimized to speed up the overall reconfiguration process.

In contrast to a static use case, developers and users of accelerators in reconfigurable GPC need to take care of the latencies caused by initialisation and deinitialisation steps of the drivers. Because these are executed during every reconfiguration cycle, they may contribute a major part to the overall reconfiguration time.

The total latency introduced by the reconfiguration steps (T_{TOT}) is an important information for scheduling decisions. If for example more than one RR is available for reconfiguration, the one with the lowest reconfiguration latency should be reconfigured first. This raises the chance of the others with higher reconfiguration overhead to be reused until they get reconfigured.

References

[1] Alan Tull, "Reprogrammable Hardware under Linux," Presentation at Embedded Linux Conference Europe 2015, Dublin, Oct. 2015. [Online]. Available: http://events.linuxfoundation.org/sites/events/files/slides/FPGAs-under-Linux-Alan-Tull-v1.00.pdf

[2] A. DeHon and J. Wawrzynek, "Reconfigurable computing: what, why, and implications for design automation," in *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, 1999, pp. 610–615.

[3] S. Wiehler and U. Langenbach, "Programming reconfigurable devices via fpga regions & device tree overlays," Presentation at 17th Free and Open Source Software Developers' European Meeting 2017, FOSDEM, Feb 2017. [Online]. Available: https://fosdem.org/2017/schedule/event/programming_reconfigurable_devs/

[4] 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. 2014. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6636314

[5] A. Ismail and L. Shannon, "FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators," in *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2011, pp. 170–177.

[6] M. Liu, Z. Lu, W. Kuehn, S. Yang, and A. Jantsch, "A Reconfigurable Design Framework for FPGA Adaptive Computing," in *2009 International Conference on Reconfigurable Computing and FPGAs*, Dec. 2009, pp. 439–444.

[7] H. K.-H. So and R. Brodersen, "A Unified Hardware/Software Runtime Environment for FPGA-based Reconfigurable Computers Using BORPH," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 2, pp. 14:1–14:28, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1331331.1331338

[8] C. Bobda, M. Majer, A. Ahmadinia, T. Haller, A. Linarth, and J. Teich, "The Erlangen Slot Machine: a highly flexible FPGA-based reconfigurable platform," in *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, Apr. 2005, pp. 319–320.

[9] 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 *2005 IEEE Conference on Emerging Technologies and Factory Automation*, vol. 2, Sep. 2005, pp. 8 pp.–338.

[10] T. Xia, J.-C. Prévotet, and F. Nouvel, "Microkernel Dedicated for Dynamic Partial Reconfiguration on ARM-FPGA Platform," *SIGBED Rev.*, vol. 11, no. 4, pp. 31–36, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2724942.2724947

[11] X. Iturbe, K. Benkrid, A. T. Erdogan, T. Arslan, M. Azkarate, I. Martinez, and A. Perez, "R3tos: A reliable reconfigurable real-time operating system," in *2010 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, Jun. 2010, pp. 99–104.

[12] D. Meyer, M. Eckert, J. Haase, and B. Klauer, "Generic operating-system support for FPGAs," in *2016 International Conference on FPGA Reconfiguration for General-Purpose Computing (FPGA4GPC)*, May 2016, pp. 7–12.

[13] K. Vipin and S. A. Fahmy, "A high speed open source controller for FPGA Partial Reconfiguration," in *2012 International Conference on Field-Programmable Technology*, Dec. 2012, pp. 61–66.

[14] S. G. Hansen, D. Koch, and J. Torresen, "High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, May 2011, pp. 174–180.

[15] K. Vipin and S. A. Fahmy, "ZyCAP: Efficient Partial Reconfiguration Management on the Xilinx Zynq," *IEEE Embedded Systems Letters*, vol. 6, no. 3, pp. 41–44, Sep. 2014. [Online]. Available: http://ieeexplore.ieee.org/document/6780588/

[16] A. Morales-Villanueva, R. Kumar, and A. Gordon-Ross, "Configuration prefetching and reuse for preemptive hardware multitasking on partially reconfigurable fpgas," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 1505–1508.

[17] G. Charitopoulos, I. Koidis, K. Papadimitriou, and D. Pnevmatikatos, "Hardware Task Scheduling for Partially Reconfigurable FPGAs," in *Applied Reconfigurable Computing*, ser. Lecture Notes in Computer Science, K. Sano, D. Soudris, M. Hübner, and P. C. Diniz, Eds. Springer International Publishing, Apr. 2015, no. 9040, pp. 487–498, dOI: 10.1007/978-3-319-16214-0_45. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-319-16214-0_45

[18] E. Lübbers and M. Platzner, "A portable abstraction layer for hardware threads," in *2008 International Conference on Field Programmable Logic and Applications*, Sep. 2008, pp. 17–22.

[19] S. Saha, A. Sarkar, and A. Chakrabarti, "Scheduling Dynamic Hard Real-Time Task Sets on Fully and Partially Reconfigurable Platforms," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 23–26, Mar. 2015.

[20] A. Ahmadinia, C. Bobda, and J. Teich, "A Dynamic Scheduling and Placement Algorithm for Reconfigurable Hardware," in *Organic and Pervasive Computing ARCS 2004*, ser. Lecture Notes in Computer Science, C. Müller-Schloer, T. Ungerer, and B. Bauer, Eds. Springer Berlin Heidelberg, Mar. 2004, no. 2981, pp. 125–139, dOI: 10.1007/978-3-540-24714-2_11. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-540-24714-2_11

[21] M. Happe, A. Traber, and A. Keller, "Preemptive Hardware Multitasking in ReconOS," in *Applied Reconfigurable Computing*, ser. Lecture Notes in Computer Science, K. Sano, D. Soudris, M. Hübner, and P. C. Diniz, Eds. Springer International Publishing, Apr. 2015, no. 9040, pp. 79–90, dOI: 10.1007/978-3-319-16214-0_7. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-319-16214-0_7

[22] K. Jozwik, H. Tomiyama, M. Edahiro, S. Honda, and H. Takada, "Comparison of Preemption Schemes for Partially Reconfigurable FPGAs," *IEEE Embedded Systems Letters*, vol. 4, no. 2, pp. 45–48, Jun. 2012.

[23] C. Kohn, "Xapp1231 partial reconfiguration of a hardware accelerator with vivado design suite for zynq-7000 ap soc processor," Mar 2015. [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp1231-partial-reconfig-hw-accelerator-vivado.pdf

[24] G. Likely and J. Boyer, "A symphony of flavours: Using the device tree to describe embedded hardware," in *Proceedings of the Ottawa Linux Symposium 2008*, vol. 2, Jul 2008, pp. 27–38. [Online]. Available: http://www.landley.net/kdocs/ols/2008/ols2008v2-pages-27-38.pdf

[25] "IEEE standard for boot (initialization configuration) firmware: Core requirements and practices." [Online]. Available: https://www.openfirmware.info/data/docs/of1275.pdf

[26] Pantelis Antoniou, "Transactional Device Tree & Overlays: Making Reconfigurable Hardware Work," San Jose, Mar. 2015. [Online]. Available: http://events.linuxfoundation.org/sites/events/files/slides/dynamic-dt-keynote-v3.pdf

[27] "UG954: Zc706 evaluation board for the zynq-7000 xc7z045 all programmable soc user guide," Mar 2016. [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf

[28] Marek Vašut, "Writing drivers for the Linux Crypto subsystem," Presentation at LinuxCon Japan 2014, Mai 2014.

[29] "UG902: Vivado Design Suite User Guide: High-Level Synthesis." [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug902-vivado-high-level-synthesis.pdf

[30] B. Conte, "crypto-algorithms," Apr. 2015. [Online]. Available: https://github.com/B-Con/crypto-algorithms/blob/master/aes.c

[31] S. Rostedt, "ftrace," 2008. [Online]. Available: https://www.kernel.org/doc/Documentation/trace/ftrace.txt

[32] "TRAPpy," 2016. [Online]. Available: https://pythonhosted.org/TRAPpy/trappy.ftrace.html

[33] "UG585: Zync-7000 all programmable soc technical reference manual," Sept 2016. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf