Accelerating Linux Bash Commands on FPGAs Using Partial Reconfiguration

Document Version
Accepted author manuscript

Link to publication record in Manchester Research Explorer

Citation for published version (APA):

Published in:
Proceedings of FPGAs for Software Programmers (FSP 2017) conference

Citing this paper
Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights
Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy
If you believe that this document breaches copyright please refer to the University of Manchester's Takedown Procedures [http://man.ac.uk/04Y6Bo] or contact uml.scholarlycommunications@manchester.ac.uk providing relevant details, so we can investigate your claim.
Accelerating Linux Bash Commands on FPGAs Using Partial Reconfiguration

Edson Horta\(^a\), Xinzi Shen\(^a\), Khoa Pham\(^a\), and Dirk Koch\(^a\)

\(^a\)School of Computer Science, The University of Manchester, United Kingdom

Abstract

The Linux Operating System is used by a wide range of companies, mainly due to its security and stability. The latest FPGA devices, with embedded processors, allow the development of SoCs running this OS. One of the key features of FPGAs is their ability to change the hardware architecture while the system is running. It is possible to reconfigure one or more regions of the FPGA while the remainder continues to work. These partially reconfigurable regions inside the FPGA can be used to implement software functions, accelerating the execution time for the system. This work presents an FPGA system that provides the required interfaces to deploy Linux bash commands mapped directly into partially reconfigurable blocks of the FPGA. These commands can be executed in any order, thanks to a hardware interface, equivalent to the software pipe interface in Linux, implemented on the static region of the FPGA. The methodology presented in this paper allows the hardware implementations of commands that can be directly interfaced to the std_in and std_out inside the FPGA. This integration is executed in a very flexible manner according to the currently called bash command. This approach can be used to improve the performance of Linux applications through the use of hardware accelerated bash commands. Moreover, this approach is providing a template that will make hardware and software integration very easy.

1 Introduction

The introduction of SoC devices containing processor cores integrated with programmable logic paved the road to develop hardware accelerators to improve the performance of pure software approaches. ARM processors running Linux OS is the de facto standard for these SoC devices. The implementation of Linux commands such as regular expression matching using hardware accelerators is an used example to improve the performance of unstructured textual data analysis [1]. These implementations focus only in the stand alone function to be accelerated. However, Linux systems are able to arrange the data processing using pipelines. Currently, a pipeline is implemented entirely in software, using buffers to store the data produced by a process and then sending it to the next process. These data transfers are managed by the OS, occupying precious process cycles from the main application.

This work proposes an scalable architecture that can implement not only Linux bash functions in hardware but also the pipes between consecutive hardware accelerators. The pipeline is implemented with a standard bus interface and the bash commands are implemented using partial reconfiguration. In order to show a proof of concept, a system containing an ARM processor running Petalinux [2] was designed with two typical bash functions, grep and translate, implemented in partially reconfigurable regions of a Zynq FPGA. The pipe interface is implemented in an AXI-Stream switch and the bash functions are connected to this switch while an AXI-Lite bus is used to access the system memories or a register file used to pass additional call arguments to the hardware modules (e.g. the compare string in case of a hardware grep command). Using drivers provided by the FPGA vendor, it is possible to send/receive data to each module separately or connecting them together in order to form a pipe command.

Our approach provides to the designer a framework that allows using pipes arbitrarily for carrying out intraprocess communication between software and hardware. Using a compatible and standardized interface allows for the development of standardized drivers, facilitating the integration effort. Using these drivers a software developer can access the hardware accelerators transparently and the OS can choose the best alternative to use, software or hardware, depending on the application requirements. As the interface between bash commands is well defined in the hardware, the development of new commands by the hardware designer is also simplified.

Another innovative feature proposed by this paper is the stand alone implementation of bash commands in hardware without having to run the synthesis, place and route tools for the static region of the FPGA more than once. The methodology used to fix the interface between the static and partial regions allows the compilation of different partial modules (bash functions) separately from the static system. Moreover, our approach allows modules to be relocated or instantiated multiple times. For example, this allows it to call two consecutive hardware grep functions by reusing the same partial bitstream. Our methodology makes use of two proprietary tools: GoAhead [3] and BITMAN [4], along with off-the-shelfvendor tools to generate the partial bitstream files.

In our flow, modules are implemented separately from the static system and when a developer wants to design a mod-
The only view will be a standardized AXI-stream interface that corresponds directly with a provided template to be used in bash (or other programs that communicate through std_in and std_out). This enormously simplifies complexity and removes the need to use a special version of the Xilinx vendor tools (i.e., different versions of the Vivado suite can be used to build the static system and the different partial accelerator modules). The result of this flow is a partial bitstream that is linked dynamically at runtime and this mechanism shares many concepts of linking software binaries, hence this concept should be very accessible also to software programmers who want to use our system.

The remainder of the paper is organized as follows: In Section 2, the pipeline technique, the hardware accelerators, and the FPGA overlay concept are reviewed. In Section 3, the system architecture is presented, along with its implementation. In Section 4, the system framework is revealed. Section 5 presents the conclusions and future work.

2 Background

2.1 Linux POSIX

The Portable Operating System Interface (POSIX) standard [5] is a commonly applied intercommunication method used by software engineers involved in the development of operating systems and/or applications based on the UNIX system. This standard aims to provide a clear interface specification for portable operating systems based on the UNIX documentation.

POSIX defines a pipeline as: "a sequence of one or more commands separated by the control operator [ ]|" (known as "pipe" in the developer's community). POSIX does not define the implementation of the commands neither the interface between them. In current operating systems, pipes are implemented as unidirectional channels and memory buffers are generally used to store the data generated by a command and the buffers are read by the next command in the pipeline.

Consequently, each command in a pipeline produces data that is used by the next command. The OS creates processes for all tasks and the synchronization is data driven. If a process has input data in its input FIFO (buffer) to process, it fires adjacent process that in turn may create data in its output FIFO which may trigger processing of the next task and so forth. This way, a sequence of pipelined bash instructions follows the model of communicating sequential processes [6].

One typical application of pipe consists in text processing which is often done by functions like grep and translate. The results from grep, a function used basically to search for patterns in a string, are written to the pipe and then the translate, a text substitution function, can read this results from the pipe and perform some text replacement.

Each command in the pipeline needs to access data from the previous one. This means that the OS has to allocate buffers and control the task of reading/writing data from/to these buffers. In addition, the OS has to implement process queues for each pipe created by the user. All these tasks include the access to physical memories in the system, creating an undesirable overlay, regarding the access time.

The proposed system uses hardware streaming ports to connect the regions containing the bash commands implemented in hardware. This approach is a hardware oriented version of the process queues mentioned before. The advantage of this method is the direct connection to the storage elements used by the hardware accelerators, increasing the efficiency of data transfers between processes.

2.2 Hardware Accelerators

Some state of the art FPGAs contain a mix of processor hard cores, memory controllers, high-speed serial links, besides the programmable logic fabric. This architecture allows the design of heterogeneous systems where it is possible to migrate software functions to hardware elements. These elements, known as hardware accelerators, are used in applications such as neural networks [7] [8], data centers [9] and scientific [10]. Although the hardware accelerators are used by software engineers, the implementation of such modules is not possible without the help of specialized hardware engineers, providing experience on RTL coding and EDA tools.

In the proposed architecture, the FPGA floorplanning, containing the static and partial regions must follow strict design rules and the interface between these two regions is the most critical part. The first approach for such interfaces was developed in 2001, when they were implemented as gaskets [11], using flops to fix their positions inside the FPGA. Another generation of partial reconfiguration flows used bus macros that are hard macros that basically consist of a pair of look-up tables (LUT) and a wire between them [12]. This hard macro was then placed in such a way that one LUT of the macro is located inside the reconfigurable region and the other LUT inside the static region. This allows implementing relocatable modules but causes a significant overhead for the macros in terms of both logic and extra delay (pass the LUTs). The latest release of the vendor tools provides an interface called partition pin [13] that fixes the position of each wire, but in a manner that cannot be fully controlled by the user. The flow is based on an incremental design method where during the static system implementation the routing is carried out to the partition pins that are located inside reconfigurable regions (which are kept otherwise empty). Then a copy of this static system is made for each permutation of region and different kind of module. After this, a place and route step is carried out for each permutation. This means a system that will provide only 5 reconfigurable regions and 10 partially reconfigurable accelerator modules, will need 50 individual (incremental) place and route steps resulting in 50 different bitstreams (1 implementation per module per region). This is necessary because the routing onto the reconfigurable regions cannot be sufficiently constrained using the vendor flow, and as a consequence, the interfaces are in general different for each region which prevents module relocation (and therefore reusing CAD-tool effort for different regions). Moreover, whenever changes are under-
taken to the static system, the routing to the partition pins inside the reconfigurable regions may likely change, which in turn requires rerunning all (in the example 50) partial module implementations. This obviously is not acceptable and an interfacing method is needed that decouples the implementation of a partial module from the implementation of the static system. This interfacing method is detailed in Section 3.

2.3 FPGA Overlay

The implementation of SoC using FPGAs is still a complex task from the software developer perspective. The hardware accelerator approach explained in the previous section turns into a more cumbersome job due to the physical placement of the hardware components inside the FPGA. In order to ease this process, it was introduced the concept of an intermediate layer hiding the hardware blocks from the software developer, analogous to a hardware driver used in PC systems. This intermediate layer, an FPGA overlay [14], configures the device with all the required hardware blocks needed by the OS and/or the application software.

This work, however, presents two important extensions to this concept: 1) We are building an FPGA overlay, the static region, with pre defined partially reconfigurable areas (overlay slots) to be used by the hardware accelerators; 2) Furthermore, we communicate through std_in/std_out POSIX primitives instead of a Python API which allows omitting Python. This is relevant as C/C++ is still dominant in embedded systems. It also contains the processor, communication buses and peripherals used to run a Linux "like" OS. The required hardware to implement the pipe concept is also present, along with all the interfaces to host a partial module inside the overlay slots. The software developer does not need to know how this infrastructure was implemented. He only must know how to call/use the available bash commands implemented in hardware.

Given a library of existing bash commands a hardware designer is able to implement them in hardware, without having to build a new system. The only aspect that must be known is the fixed hardware interface used to communicate with the hardware module (bash command). With this information it is possible to run the EDA tools independently from the static system (FPGA overlay) implementation and generate the required bash command in the format of partial bitstreams. A bitstream is a configuration file containing the information about how the FPGA must connect/program its resources to implement a specific hardware architecture inside the FPGA. Whereas a partial bitstream contains the same information, but related to a specific region of the device.

3 System Architecture

The proposed system implementing hardware bash commands and pipe interface is shown in Figure 1. The communication between software and hardware is carried out through wrappers commands that create buffers in memories and call a driver to carry out the communication. The hardware accelerators access the DDR memory through a DMA engine connected to the AXI-Stream switch. The AXI-Lite interface is used to control the hardware accelerators.

The AXI-Stream switch is used as the physical implementation of the software pipe interface. When only one command is used at a time, the switch connects the specific accelerator directly to the DMA controller. When a pipe is used, the switch connects the output from the first (CMD0) with the input of the second accelerator (CMD1) module. In this configuration, the data injected to the first command and the data produced by the second command comes from the DMA engine also.

As seen on Figure 1, the system is divided in one static and two partial regions (0 and 1), used to host the commands that will be accelerated through the FPGA resources. The interface between each partial and the static region is the same, facilitating the development of new hardware accelerators independently from the static system.

3.1 HW Implementation

The static region (FPGA overlay) contains the logic that will not change during the system execution and in our case this comprises the ARM processor, the DMA controller, the AXI-Lite and the AXI-Stream interface. The
DDR memory controller is embedded in the ARM core inside the FPGA. There are two partial regions (overlay slots) allocated to host up to two command accelerators.

The system floorplanning and partial bitstream generation are done with off-the-shelf commercial tools, GoAhead and BITMAN. These tools are used because the partial design flow provided by the FPGA vendors is not sufficient to: 1) fix the interface wires in the required positions using a regular routing structure; 2) implement only the partial module and extract its correspondent partial bitstream.

GoAhead is a tool for implementing partially reconfigurable systems on Xilinx FPGAs with capabilities that are not available when using the vendor partial reconfiguration flow. The most important capabilities that will be needed in our system are that partially reconfigurable modules (Linux bash commands) can be relocated and instantiated multiple times within the same system.

The GoAhead tool is used for the static region implementation to prevent the vendor tools from using logic and routing inside the partial reconfigurable regions, similar to what is done in [15] and [16]. In the partial module implementation, GoAhead prevents the use of logic and routing resources around the partial reconfigurable region (basically implementing a fence around a partial module for place and route). The tool is also capable of constructing a routing path inside this fence in order to connect the static and partial regions of the designs.

### 3.1.1 Static Region

The methodology used to implement the static region starts with a GoAhead script file template that defines the position of the module areas for the reconfigurable blocks. The script further defines the interfaces between static and partial regions and then the GoAhead tool utilizes it to generate various placement and routing constraints. By using the description of the static modules plus the constraints generated by GoAhead, the FPGA vendor’s tools can then run logic synthesis followed by the physical implementation all the way down to the generation of the static region full bitstream. This full bitstream for the static region is then used to generate the boot image for the system.

Figure 2 shows our static system implemented on a Zynq 7000 device. It is possible to identify on the right side of Figure 2 two empty partial regions reserved to host the Linux bash commands (partial 0 and partial 1). The zooming windows show the routing signals (horizontal lines, in white) that connects the static system with the hardware accelerators in the partial regions 0 and 1. As can be seen in Figure 2, the routing signals follow a regular pattern in the FPGA fabric, different from the implementation through the partial reconfiguration design flow available from the commercial tools. These regular patterned wires act like a hardware socket to plug in the hardware accelerated modules independently generated by the implementation tools.

### 3.1.2 Partial Region

The implementation of each bash command is done through another script file for the GoAhead tool. This script defines the position of the module and its interface, in the same locations reserved in the static design implementation.

Figure 3 shows the implementation of a partially reconfigurable module with the grep command inside the upper partial region of the FPGA. The size of the bounding box and the interface position match to those from the static system. Similarly, Figure 4 shows the implementation of a partially reconfigurable module with the translate command inside the bottom partial region of the FPGA.

The zooming windows in Figures 3 and 4 show the routing signals used to connect the hardware accelerated modules to the static system. The pattern shown on both designs, for the white horizontal lines, is the same generated for
Figure 5 Module Duplication: CMD1 -> CMD1.

the static signals, providing the exact matching for all the logical interface signals, regarding their physical positions inside the device.

After using GoAhead, the vendor’s tools generate full bitstream files containing: 1) the first bash command; 2) the second bash command. These full bitstreams must be manipulated to generate the partial bitstreams to configure only the required logic and routing resources from the FPGA. This manipulation involves the acquisition of information about how the bitstream maps all the resources inside the FPGA fabric and how to modify this mapping. The information about the bitstream resource mapping allows the definition of a specific area inside the fabric and the generation of the corresponding partial bitstream to configure it. BITMAN is the tool used to provide this functionality, generating a partial bitstream for each bash command. Another feature the tool provides, not present in the commercial tools, is the relocation of partially reconfigurable modules of the FPGA fabric to a new region, provided that they use the same interface. In our system, this feature can be used, for example, to execute two identical commands through the pipeline, as illustrated by Figure 5.

4 System Framework

The framework provided comprises two environments, for different target users: 1) software developer and 2) hardware accelerator developer. The former is responsible for the development of applications running on the OS and the latter generates the hardware accelerated bash commands. The first environment uses the full bitstream, containing the FPGA overlay, and the SDK (Software Development Kit) from Xilinx to generate the drivers for the hardware IPs (bash commands) and the AXI bus interfaces. The operating system is then generated through the use of Petalinux tools and the drivers from SDK. The image generated by Petalinux is then loaded into an external memory and used to initialize the SoC. The bash commands present in the overlay slots are accessed by the OS using wrapper functions that call the IP drivers created by SDK.

Four device drivers have been generated to access the hardware components through the operating system:

1. simple-dma: used to access the external memories directly
2. uio0: used to access the AXI-Stream switch
3. uio1: used to access the access register in partial region 0
4. uio2: used to access the access register in partial region 1

The software developer needs to know only the wrapper command to call the hardware accelerators. This feature simplifies the integration of these new “commands” into the application.

The second environment focuses on the implementation of each bash command in hardware, independently from the static design. After validating the hardware module with stand alone simulations, the hardware designer follows the steps described in Section 3.1.2 and generates the partial bitstream for each module.

After booting the SoC with the image generated by Petalinux, the system is able to run the regular Linux bash commands, but not the hardware accelerators, due to the partial regions being empty. The modules can be downloaded to the FPGA using the JTAG port and then it is necessary to load the drivers to access them and the AXI-Stream switch (uio_pdrv_genirq), along with the driver for the DMA engine (simple-dma).

As an example, assuming that grep was downloaded to partial region 0 and translate was downloaded to partial region 1, the following wrapper commands are available:

- grep_acc <pattern> FILE
- trans_acc <SET1> <SET2> FILE
- trans_acc -d <SET1> FILE
- grep_trans_acc <pattern> <SET1> <SET2> FILE
- grep_trans_acc <pattern> -d <SET1> FILE

The last two are pipeline commands to execute grep followed by translate (to substitute or delete characters).

5 HLS Application

The proposed system can also be used to implement functions generated by High Level Synthesis tools, such as Vivado HLS. In [17], a partially reconfigurable system containing three video filter algorithms is presented. Each algorithm is generated by Vivado HLS and then implemented as partial modules that can be reconfigured on the fly according to the video features required. Using the proposed framework described in the previous section, a bash command containing any one of the video filter algorithms can be implemented in an overlay slot. Figure 6 shows the implementation of the Sobel video filter using the RTL description generated by Vivado HLS.

6 Conclusions and future work

This paper presents a different approach for FPGA overlays, adding the concept of overlay slots, partially reconfigurable regions used to host hardware accelerators. The proposed overlay can be modified to address different numbers
of accelerated commands in the FPGA or different pipe interfaces implemented in hardware, although this is not the main purpose of the system.

A two-environment framework was presented, focusing on the software and hardware designers. The former is used to provide hardware accelerators and hide all the aspects related to the use of the EDA Tools. The latter makes possible to independently generate hardware accelerators confined in the overlay slots, without worrying about the development of complex drivers to access them.

An SoC was implemented with two bash commands transformed in hardware accelerators. The static system design needs to be done only once and the hardware accelerated modules are implemented independently from the static system. Both implementations do not need to use the partial reconfiguration flow from the available commercial tools.

A future work derived from this paper could be the implementation of self-adaptive systems, according to the different applications running at the moment. These systems could dynamically choose to execute a bash command from a pure software implementation or from a partial bitstream generated by our methodology.

7 Acknowledgment

This work is supported by the European Commission under the H2020 Programme and the ECOSCALE project (grant agreement 671632).

In addition we gratefully thank Xilinx for supporting our work by providing us with tools and development boards donations.

8 Literature


