Maturating an operating system for single board system-cubieboard

[1]SATHISH.M, [2]BOOPATHY.P

[1]M.Tech (CSE), PRIST University, India, sathishvasant@gmail.com

[2]Asst Prof, Dept of CSE, PRIST University, India, pandiboopathy@gmail.com,

Abstract— The advances in the programmable hardware have lead to new architectures, where the hardware lead dynamically adapted to the application to gain better performance. One of the many challenging problems in realizing a general purpose reconfigurable system is the placement of the modules on the reconfigurable functional unit (RFU). Reconfigurable computing systems still lack an established OS foundation that covers both software and hardware parts. This implies that tomorrow's applications will make use of both the instruction set processor (ISP) and the reconfigurable logic in order to provide the user with maximum performance.

In this paper we present , The design and implementation of a Monolithic-Kernel Single Board Computer (SBC) - Cubieboard GNU/Linux-like operating system on ARM (Advanced RISC Machine) platform in technical details, including boot loader design - UBOOT, building the Kernel - uImage, design of root file system and init process. The Single Board Computer Operating System (SBC OS) is developed on Linux platform with GNU tool chain.

Keywords— Single board computer, UBOOT, ARM, UImage, Cubieboard, Monolithic Kernel, Init Process.

INTRODUCTION

RECONFIGURABLE computing (or RC), as a discipline, has now been in existence for well over a decade. During this time, significant strides have been made in fabrication that are now providing hybrid computer processing unit (CPU)/field programmable gate array (FPGA) components with millions of free logic gates, as well as diffused intellectual property (IP) in the form of high-speed multipliers and SRAM blocks Unfortunately, researchers have thus far struggled to develop tools and programming environments that allow programmers and system designers—not just hardware designers—to tap the full potential of the new reconfigurable chips [9]. This deficiency is in part due to the absence of modern operating system and middleware services that extend across the CPU/FPGA boundary. These layers, along with a high-level language, form an abstract computational model of a virtual machine. Importantly, these layers provide the concurrency and synchronization mechanisms used within modern software concurrency models such as asynchronous threads

In this paper, we first outline and discuss the issues of currently accepted computational models for hybrid CPU/FPGA systems. We then discuss the need to adopt a modern virtual machine approach and associated abstract computational model, which hides the platform specific CPU/FPGA distinctions from the programmer.

We then present Monolithic-Kernel Single Board Computer GNU/Linux like operating system on ARM platform. The advantage of the system is described in the following.

A. Monolithic-Kernel Architecture

Unlike micro-kernel in MINIX operating system which slower processing system due to additional message passing, the SBC OS is designed as a Monolithic-Kernel analogous to the famous GNU/Linux. With such kind of architecture, the faster processing, the modularity and structure can be improved significantly therefore is suitable for Single Board Computers.

B. For both Single Board Computer Development and Curriculum Teaching

On one hand, the essential techniques related to operating systems and ARM machines are involved, e.g., boot loader design - UBOOT, building the Kernel - uImage, design of root file system and init process. All of these are obviously helpful for development

on ARM based Single Board Computer as well as for students to learn and study [2]. On the other hand, the SBC OS is designed more readable, of which the source codes can be provided to students, guiding them to design tiny Single Board Computer operating system on ARM platform from scratch.

C. Modularity and Structure

Each functionality should be found in a separate module, and the file layout of the paper should reflect this. Depending on their function, many capabilities can also be built into optional, runtime-loadable, modular components [8]. These can be loaded later when the particular capability is required. Within each module, complex functionality is subdivided in an adequate number of independent functions. These (simpler) functions are used in combination to achieve the same complex end-result.

RELATED WORK

The operating system ReconOS [2] extends the multithreaded programming model to the domain of reconfigurable hardware. Instead of regarding hardware modules as passive coprocessors to the system CPU, they are treated as independent hardware threads on an equal footing with software threads running in the system. In particular, ReconOS allows hardware threads to use the same operating system services for communication and synchronization as software threads, providing a transparent programming model across the hardware/software boundary. This transparency makes the design space exploration regarding the hardware/software partitioning of an application a straightforward task and facilitates self-adaptation. Since all threads use identical programming model primitives such as semaphores, mailboxes or shared memory, they do not need to know whether their communication peers are software threads executed on the CPU or HW threads mapped to the reconfigurable fabric. Thus, the HW/ SW partitioning of an application can be changed at design or run-time by simply instantiating the appropriate threads. ReconOS supports dynamic reconfiguration of hardware threads by taking advantage of the partial reconfiguration capabilities of Xilinx FPGAs.

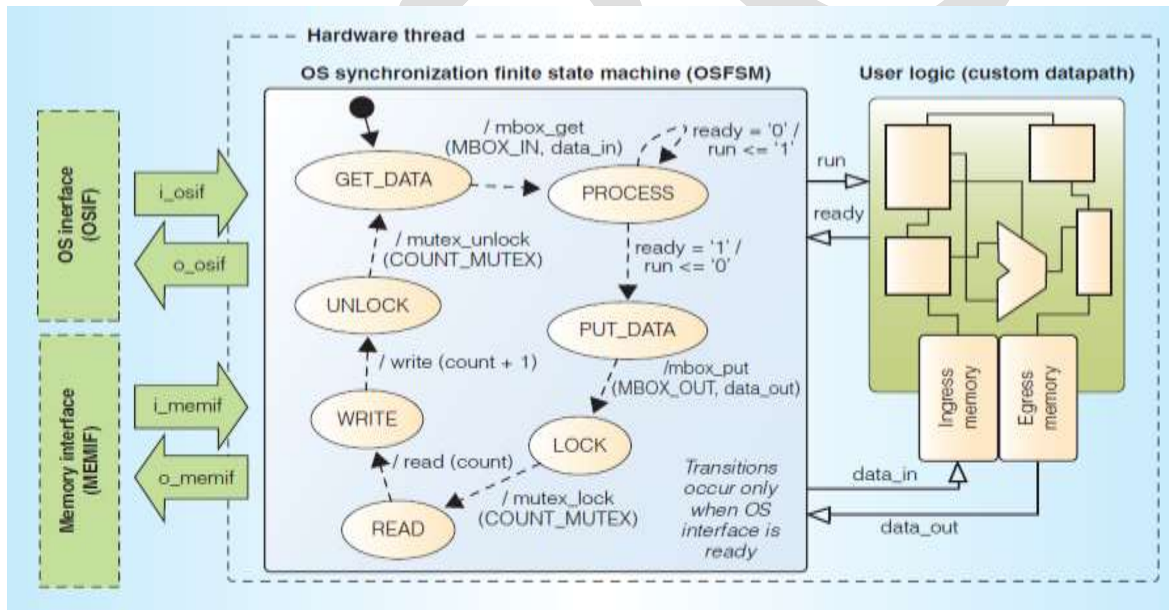

Fig. 1 A ReconOS hardware thread comprises the OS synchronization finite state machine and the user logic implementing the data path. Together with the OS interface (OSIF), the OS synchronization finite state machine enables seamless OS calls from within the hardware thread. The memory interface (MEMIF) provides the hardware thread with access to the ReconOS memory subsystem.

Basic Programming Architectures

Before introducing new architecture variants, we must first consider existing systems [3]. Traditional FPGA structures have primarily been serially programmed single-context devices, allowing only one configuration to be loaded at a time. This type of Field Programmable Gate Arrays (FPGA) is programmed using a serial stream of configuration information, requiring a full reconfiguration if any change is required. Designers of reconfigurable systems have found this style of configuration to be too limiting to efficiently implement run-time reconfigurable systems.

In some cases, configurations do not occupy the full reconfigurable hardware, or only a part of a configuration requires modification. In both of these situations a partial reconfiguration of the array is desired, rather than the full reconfiguration supported by the serial device mentioned above. In a partially reconfigurable FPGA, the underlying programming layer operates like a RAM device. Using addresses to specify the target location of the configuration data allows for selective reconfiguration of the array. Frequently, the undisturbed portions of the array may continue execution, allowing the overlap of computation with reconfiguration [10]. When configurations do not require the entire area available within the array, a number of different configurations may be loaded into otherwise unused areas of the hardware. Partially run-time reconfigurable architectures can allow for complete reconfiguration flexibility such as the Xilinx 6200 [Xilinx96], or may require an full column of configuration information to be reconfigured at once, as in the Xilinx Virtex FPGA [Xilinx99].

In contrast, a multi-context FPGA includes multiple memory bits for each programming bit location. These memory bits can be thought of as multiple planes of configuration information [DeHon, Trimberger]. Only one plane of configuration information can be active at a given moment, but the device can quickly switch between different planes, or contexts, of already-programmed configurations [11]. In this manner, the multi-context device can be considered a multiplexed set of single context devices, which requires that a context be fully reprogrammed to perform any modification to the configuration data. However, this requires a great deal more area than the other structures, given that there must be as many storage units per programming location as there are contexts.

SYSTEM ARCHITECTURE – SINGLE BOARD COMPUTER SYSTEM CUBIEBOARD

The SBC OS normally reside in large-capacity devices such as hard disks, CD-ROMs, USB disks, network servers, and other permanent storage media is shown in Fig.2 [4]. When the processor is powered on, the memory does not hold an operating system, so special software is needed to bring the SBC OS into memory from the media on which it resides. This software is normally a small piece of code called the boot loader. On a desktop PC, the boot loader resides on the master boot record (MBR) of the hard drive and is executed after the PC's basic input output system (BIOS) performs system initialization tasks.

- Kernel 3.4.105 with broad hardware support, headers and some firmware included
- Ethernet adapter with DHCP and SSH server ready on default port (22) with regenerated keys
- Enabled audio devices: analog, HDMI, spdif and I2S.
- Advanced IR driver with RAW RX and TX (disabled by default / you need an IR diode)
- PWM ready on pin PB2 (Cubietruck)
- Bluetooth ready (working with on-board device / disabled by default – insserv brcm40183-patch)
- I2C ready and tested with small 16×2 LCD. Basic i2c tools included.
- SPI ready and tested with ILI9341 based 2.4″ TFT LCD display.
- Drivers for small TFT LCD display modules.

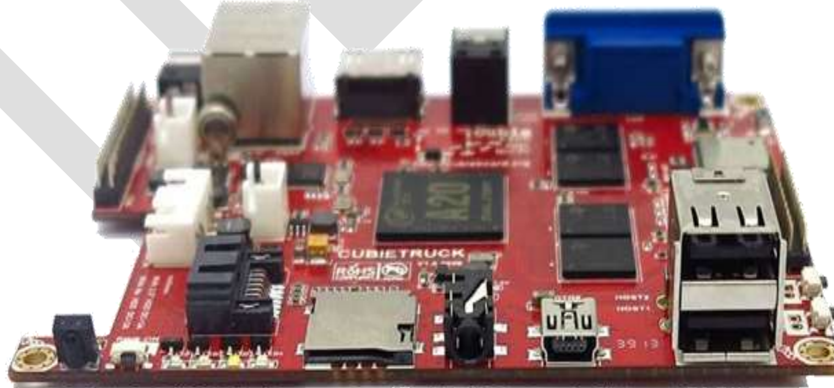

Fig. 2: Single Board Computer – Cubieboard

Performance tweaks:

- /tmp & /log = RAM, ramlog app saves logs to disk daily and on shut-down (ramlog is replaced with busybox-syslogd on Jessie)
- IO scheduler NOOP for SD, CFQ for sda (mechanical hard drive). (change in /etc/sysfs.conf)
- journal data writeback enabled. (/etc/fstab)
- commit=600 to flush data to the disk every 10 minutes (/etc/fstab)
- optimized CPU frequency scaling 480-1010Mhz with interactive governor (/etc/init.d/cpufrequtils)
- eth0 interrupts are using dedicated core

Limitations

- Some drivers compile successfully some displays fatal error: mach/sys_config.h: No such file or directory. Currently no idea how to fix this.
- On board Bluetooth firmware loading sometime fails. Reboot helps.
- NAND install script sometime fails. Dirty but working workaround – installing Lubuntu to NAND with Phoenix tools and run the nand-install again.
- Gigabit ethernet transfer rate is around 50% of its theoretical max rate (hardware or firmware issue)
- Shutdown, reboot and battery troubles regarding poor AXP chip driver (fixed in mainline kernel).
- Due to bad PCB placement, there is some crosstalk between Wifi and VGA in certain videomodes.
- No serial console under Jessie & Ubuntu
- No LIRC under Ubuntu

PERFORMANCE OF THE MONOLITHIC KERNEL OPERATING SYSTEM

Cubieboard is Single Board Computer, is the 3rd board of Cubieteam, also name it Cubieboard3 [5]. It's a new PCB model adopted with Allwinner A20 main chip, just like Cubieboard2. But it is enhanced with some features, such as 2GB memory, VGA display interface on-board, 1000M nic, WIFI+BT on-board, support Libattery and RTC, SPDIF audio interface.

A) Building and Configuring the Kernel

The Kernel is responsible for managing the bare hardware within our chosen target system. It takes care of scheduling use of the available hardware resources within a particular SBC OS. Resources managed by the Kernel include system processor time given to programs, use of available RAM, and indirect access to a multitude of hardware devices—including those customs to our chosen target [7]. Kernel configuration allows us to add and remove the peripheral devices. Depending on their function, many capabilities can also be built into optional, runtime-loadable, modular components.

We need to configure the options that are needed to have it in our Kernel before building it. The target is to have an appropriate .config file in our Kernel source distribution. Depending on our target, the option menus available will change, as will their content [6]. Some options, however, will be available no matter which embedded architecture we choose. After the environmental setup, make menuconfig runs a text-based menu interface

B) File structure Hierarchy

The most operations conducted by the Linux Kernel during system startup is mounting the root file structure is shown in Fig.3. The Linux Kernel itself doesn't dictate any file system structure, but user space applications do expect to find files with specific names in specific directory structures [12]. Therefore, it is useful to follow the de facto standards that have emerged in Linux systems.

Each of the top-level directories in the root file system has a specific purpose. Many of these, however, are meaningful only in multiuser systems in which a system administrator is in charge of many servers or workstations employed by different users. In most embedded Linux systems, where there are no users and no administrators, the rules for building a root file system can be loosely interpreted.

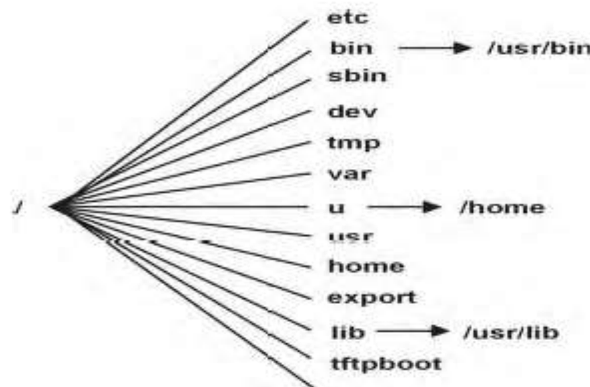

Fig. 3: Structure of file system

C) Init Process and Run levels

In conventional Linux systems, init is the first process started when a Linux Kernel boots, and it is the ancestor of all processes [5]. Its primary role is to start appropriate service processes for the "state" the system is to run in at boot and to shutdown/start appropriate services if the system state changes (such as changing to the halt/shutdown state). It can also create consoles and respond to certain types of events.

Init's behavior is determined by its configuration file /etc/inittab. Lines in /etc/inittab have the following syntax: id:runlevels:action:process

where:

id	—	1–4 (usually 2) character name for the line, totally arbitrary;
runlevels	—	a list of runlevels the line applies to;
action	—	what init is to do and/or under what conditions;
process	—	program/command to be run.

The ID for a line is completely arbitrary though there are some conventions for standard lines. For example, the IDs for lines that start services are "ln" where n is the runlevel. UNIX had runlevels 0–6, and these are the most commonly used runlevels, though Linux can use 0–9. The runlevel 1 can also be denoted by "s" (on some systems s may cause password checking that 1 does not). The runlevels component of an inittab line can be a list of runlevels. E.g., 123 would mean the line applies to runlevels 1, 2, and 3.

CONCLUSION

In this paper, we have presented single board computer operating system, which framework embrace, ARM based Monolithic kernel Operating System. Kernel is one of the most integrating areas of computer operating systems, Many kernel approaches have been proposed in previous sessions even though Monolithic kernel Operating System provide good performance than traditional

methods and are very much suitable to single board computer system. Our experience in ARM based Monolithic kernel operating system shows that these features can significantly higher the entry barrier for reconfigurable computing technology.

REFERENCES:

[1] Andreas Agne, Marco Platzner, Enno L¨ubbers "Memory Virtualization for Multithreaded Reconfigurable Hardware" 2011 21st International Conference on Field Programmable Logic and Applications

[2] Markus Happe • Enno Lu¨bbers • Marco Platzner "A self-adaptive heterogeneous multi-core architecture for embedded real-time video object tracking ", J Real-Time Image Proc (2013) 8:95–110

[3] Katherine Compton, Zhiyuan Li, James Cooley, Stephen Knol "Configuration Relocation and Defragmentation for Run-Time Reconfigurable Computing" IEEE Transactions on VLSI Systems, 2002

[4] Cubieboard / Cubietruck Debian Wheezy SD Card image, 2013, (http://www.igorpecovnik.com/2013/12/24/cubietruck-debianwheezy- sd-card-image).

[5] S.Pravin Kumar#1, G.Pradeep#2, G.Nantha Kumar#3, C.Dhivya Devi "Developing an ARM based GNU/Linux Operating System for Single Board Computer – Cubietruck" International Journal of Engineering and Technology (IJET)

[6] V. Nollet et al., "Designing an Operating System for a Heterogeneous Reconfigurable SoC," Proc. 17th Int'l Symp. Parallel and Distributed Processing, 2003, doi:10.1109/ IPDPS.2003.1213320.

[7] A. Agne, M. Platzner, and E. Lu¨ bbers, "Memory Virtualization for Multithreaded Reconfigurable Hardware," Proc. Int'l Conf. Field Programmable Logic and Applications (FPL 11), 2011, pp. 185-188.

[8] D. Andrews et al., "Achieving Programming Model Abstractions for Reconfigurable Computing," IEEE Trans. Very Large Scale Integration Systems, vol. 16, no. 1, 2008, pp. 34-44.

[9] Y. Wang et al., "A Partially Reconfigurable Architecture Supporting Hardware Threads," Proc. Int'l Conf. Field-Programmable Technology (FPT 12), 2012, pp. 269-276.

[10] K. Bazargan, R. Kastner, and M. Sarrafzadeh, "Fast Template Placement for Reconfigurable Computing Systems," IEEE Design and Test of Computers, vol. 17, no. 1, 2000, pp. 68-83.

[11] E. Lu¨bbers andM. Platzner, "CooperativeMultithreading in Dynamically Reconfigurable Systems," Proc. Int'l Conf. Field Programmable Logic and Applications (FPL 09), 2009, pp. 551-554.

[12] M. Happe, E. Lu¨ bbers, and M. Platzner, "A Self-Adaptive Heterogeneous Multi-core Architecture for Embedded Real-Time Video Object Tracking," J. Real-Time Image Processing, vol. 8, no. 1, 2013, pp. 95-110