Homayoun

Reference 8

Asymmetric Multi-Processor Architecture for Reconfigurable System-on-Chip and Operating System Abstractions

Xin Xie, John Williams, Neil Bergmann
School of Information Technology and Electrical Engineering,
The University of Queensland
Brisbane, Australia
{xxie; jwilliams; n.bergmann}@itee.uq.edu.au

***Abstract*— We propose an asymmetric multi-processor reconfigurable SoC architecture comprised of a master CPU running embedded Linux and loosely-coupled slave CPUs executing dedicated software processes. The slave processes are mapped into the host OS as ghost processes, and are able to communicate with each other and the master via standard operating system communication abstractions. Custom hardware accelerators can be also added to the slave or master CPUs. We describe an architectural case study of an MP3 decoding application of 12 different single and multi-CPU configurations, with and without custom hardware. Analysis of system performance under master CPU load (computation and IO), and a Time-Area cost model reveals the counter-intuitive result that multiple CPUs and appropriate software partitioning can lead to more efficient and load-resilient architecture than a single CPU with custom hardware offload capabilities, at a lower design cost.**

I. Introduction

Modern embedded systems are increasingly required to meet the competing requirements of real-time or near real-time performance, while satisfying tight time-to-market and interoperability requirements. Real-time performance is typically offered by dedicated custom hardware acceleration or microprocessor running specific firmware or microkernel, but the rapid development and interoperability requirements are more readily provided by commonly available operating systems (OS) such as embedded Linux.

One approach to meet these requirements is to virtualize the OS by running it as a low priority process on top of a real-time kernel. This approach is complicated, error-prone and requires an ongoing maintenance and porting effort. Another is to use custom hardware to accelerate critical sections of an application. Custom hardware design increases non-recurring costs and software/hardware co-design complexity.

Instead of multiplexing multiple software environments onto a single CPU or using custom hardware, we propose an asymmetric, reconfigurable System-on-Chip (SoC) multi-processor architecture, implemented on commodity FPGA resources. The proposed approach employs multiple CPUs, dual-port on-chip memory and FIFO type communication links. This architecture is expected to achieve the low memory latency environment for parallel processes execution, in a generic platform suitable for a wide variety of applications.

The major challenge of implementing asymmetric multiprocessor architecture (ASMP) suitable for SoC is the programming model – how can the hardware resources be exposed to the developer in a productive way without compromising performance? Our approach is to represent processes running on slave CPUs as processes in the central (master) operating system. Communication between ghost processes and other software processes on the master CPU is achieved transparently by extending the standard Unix/Linux process FIFO/pipe Inter-Process Communication (IPC) concept.

The main contribution of this research is to provide a software framework based on existing OS suitable for FPGA-based multiprocessor architecture. This paper presents a detailed description of the operating system integration model, as well as a comprehensive architectural case study. The reconfigurability of FPGAs is not used in a run-time sense, but rather as a flexible implementation fabric allowing the system architecture to be readily mapped to the application at hand.

The paper is organised as follows: a review of existing multi-processor SoC related research and techniques is presented in Section II, Section III describes the hardware subsystems and their interconnection, while Section IV introduces the OS integration model. Section V is a comprehensive case study of the architecture applied to an MP3 decoding application which is accompanied by analysis and discussions. Lastly, Section VI presents our conclusions and future research plan.

II. Background

A. *Acceleration techniques for FPGA based SoC*

One technique to improve performance on FPGA based SoC is to use customized processor cores, e.g. adding a reactive Instruction Set Architecture (ISA) and direct signal lines to existing CPUs [1]. However this technique requires modifications to the compiler to recognize the new features. It is also rare that a sufficiently large portion of the task can be mapped into a limited custom opcode set with fixed instruction format, to escape the implications of Amdahl's law.

A more common approach in the FPGA community is to design a custom hardware core to replace the critical section of a specific application, while the CPU carries other less-critical sections. However, this approach requires significant efforts in hardware/software co-design. At the same time, there is generally no kernel level OS support for the custom hardware, which can result in portability and programmability problems on the platforms other than the one initially implemented.

From the software perspective, a real-time OS or microkernel can be used on an FPGA based SoC, which offers guarantees on metrics such as interrupt latency. Operating systems such as Linux do not offer such hard real-time performance natively, but instead require significant modifications or extensions. Examples of this approach include RTLinux [2] and RTAI [3]. Porting such microkernel OS attracts non-trivial software development cost which can be unsuitable and unworthy especially in the case of hardware platform changes in the future.

B. *Multiprocessors on FPGAs*

The most common multiple processor implementations are based on a Symmetric Multi-Processor (SMP) architecture as widely used in workstation and server environments for High Performance Computation (HPC). Similar to the conventional SMP system, embedded SMP systems require OS integration [4] and hardware cache management [5]; both can be costly for an embedded system. Coherent caches are particularly expensive to implement in commodity FPGA architectures. In real terms the performance improvement in SMP is likely to be marginal due to OS associated overhead and the growing gap between processor speeds and memory latency [6, 7].

SMP is a one-size-fits-all approach, appropriate for fixed silicon when a designer must hedge their bets against all possible future uses of the architecture. FPGAs on the other hand achieve their potential only when the system architecture is created to match the application. ASMP systems can dedicate the required computing resources to computing intensive tasks, and distributed memories for each CPU avoid memory and bus bandwidth scalability issues. Existing (single CPU) operating systems can be used on ASMP system with less OS kernel changes (avoiding issues in IRQ/load balancing, cache coherence etc.)

A related asymmetric system was proposed with a master CPU and up-to eight co-processors connected with FIFOs [8]. The OS level integration of the hardware FIFO was inspirational for this research. However the co-processor is an 8-bit microcontroller programmed in assembly language, and its performance was such that the architecture was better suited to real time control applications than high speed data processing.

C. *MicroBlaze and Linux*

The hardware is based on primarily vendor-provided IPs including the Xilinx MicroBlaze soft-core CPU, hardware FIFO and on-chip dual-port memory primitives. MicroBlaze is a 32 bit RISC type soft-core CPU from Xilinx and takes on the role as both master and slave CPUs in the proposed architecture. The CPUs can be customized in different areas, including bus interfaces, cache size and hardware multiplier support [9]. At the same time, it has three different bus interfaces: Local Memory Bus (LMB) for the fast on-chip memory accessing, On-chip Peripheral Bus (OPB) for the various shared on-chip peripheral, and the Fast Simplex Bus (FSL) for the direct connection to the peripherals or processors

uClinux [10] is used as the OS on the master CPU. uClinux refers to a configuration of the regular Linux kernel that supports CPU architectures lacking hardware memory management support (such as the MicroBlaze). Many existing application are available from Linux that is readily available for the development environment. Running conventional OS on FPGA based SoC is a realistic approach that can utilize existing software methodology while retaining the advantage of the custom computing platform.

III. Hardware Architecture for FPGA based ASMP

This section contains an overview of the hardware architecture, while more complete details can be found in our earlier work [11]. The architecture is designed with generality, ease of use and performance as its primary objectives. Figure 1 shows an overview.

A. *Master CPU subsystem*

The Master CPU subsystem (left-most region of Fig. 1) is a fairly typical Linux-capable MicroBlaze system. Off-chip memory is required due to the memory footprint of the Linux kernel and the file system, which implies the use of CPU caches to achieve reasonable performance.

The master CPU communicates with slave CPUs through the specific FIFO connections to support the intended applications. In addition, the master CPU can also control the running state of the slave CPUs and reprogram the slave CPUs at run time as described in Section III.

B. *Slave CPU subsystem*

The primary motivation of having multiple slave CPUs is to benefit from parallel process execution, and to provide a dedicated environment for those computing intensive process. Typical SoC systems running applications inside the OS have the bottleneck of operating system overheads and non-deterministic memory caches. By giving slave CPUs their own low-latency local memory busses and running an exclusive single process, we can avoid the cost of an operating system for these tasks. The basic structure of the slave CPU subsystems is illustrated in the lower-right region of Fig. 1.

We utilize MicroBlaze's Harvard architecture and attach separate dual-port memories to the instruction and data memory interfaces. The other port of each dual-port memory is connected to the global bus which can be accessed from master MicroBlaze. By keeping the main data transport on the application-specific FIFO network, and only infrequent code updates on the shared global bus, bus traffic is significantly reduced.

By executing from fast on-chip memory, slave CPUs do not require instruction or data caches. In contrast to the 7-10 cycles required for external memory accesses, slave CPUs have fast (1-2 cycle) and predictable memory accesses, ideal for real-time tasks.

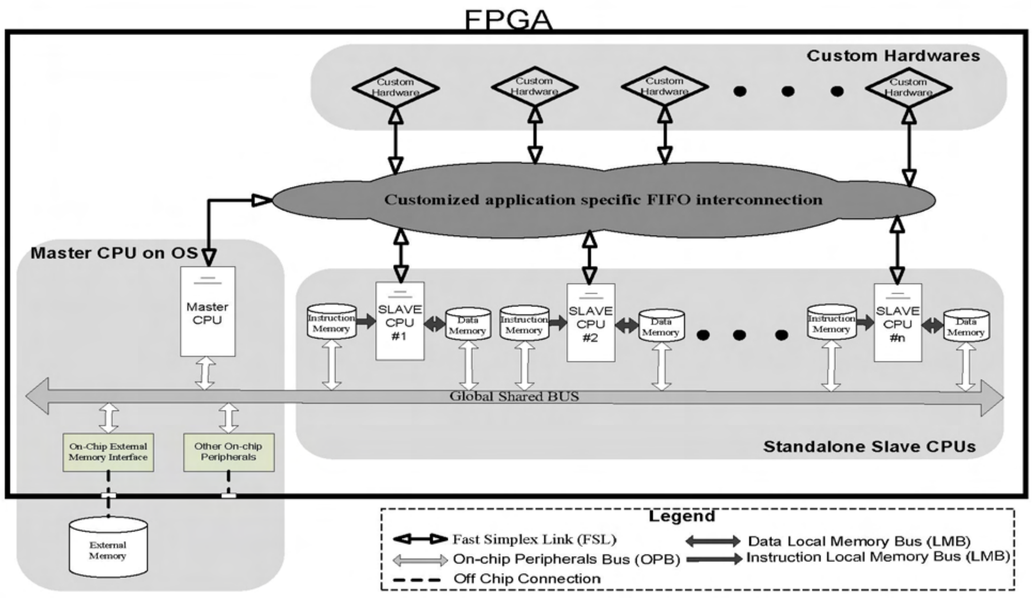

Fig. 1. FPGA base ASMP architecture

A total of eight slave CPUs can be connected to master CPU directly, a limitation imposed by the number of FSL ports on the MicroBlaze architecture. However, if not all slave CPUs require a direct data connection to the master, then the number of CPUs is limited primarily by the available logic resources. Table I shows the resource usage for a single slave CPU with 8KB each of on-chip instruction and data memory (Xilinx Virtex4 LX25 device). This on-chip memory is potentially the strongest limiting factor, however, the amount of memory allocated to each CPU can be carefully tuned to match the execution process that is will host.

The software environment of the slave CPUs is kept deliberately simple, with no support for multithreading or making master OS system calls. The current intention is to offload primarily computational tasks, with high speed data IO requirements.

C. *Integration of slave CPUs to master CPU*

An application-specific architecture is created by using the connecting these CPUs with hardware FIFOs in a network topology that matches the data flow requirements of the application (upper right, Fig. 1). The Xilinx FSL bus is ideal for this purpose. To send data to another CPU, it is written out on the appropriate FSL port. FSL is also used to attach any hardware accelerator units to CPUs that require them.

To reprogram a slave CPU, the master can write directly into its code and data memories. However, for predictable systems the ability to halt and reset these slaves is required. A controller (not shown in Fig. 1) was created for that purpose, which can manage up to 32 slave CPUs. Reprogramming a slave CPU is achieved by the following steps: (1) master CPU transmit a halt signal to the slave CPU, (2) transfer the slave CPU program code into the corresponding slave CPU's data and instruction memory from the external storage device, and (3) clear the halt signal of the slave CPU.

IV. OS INTEGRATION FOR FPGA BASED ASMP

The software abstraction of communications channel in the proposed FPGA based ASMP is critical to the system programmability and performance. The approach adopted is to extend the Unix/Linux software FIFO implementation to the ASMP communication channel.

A. *Utilities for Slave CPU control*

Section III.C described the methods used to reprogram and

TABLE I. SLAVE MICROBLAZE FPGA RESOURCES USAGE

Selected Device: Virtex-4 LX25		
Number of Slices:	1332 out of 10752	12%
Number of Slice Flip Flops:	724 out 21504	3%
Number of 4 input LUTs:	1822 out 21504	9%
Number of BRAMs*:	8 out of 72	20%
*used as 8kB+8kB I/D memory for application		

TABLE II. UTILITY SOFTWARE

Pause slave CPU # slave_control –pause slave1
Start slave CPU # slave_control –start slave1
Read back the slave CPU memory and save it to an output memory image # slave _control –read slave1 > slave1_image
Write back the slave CPU memory from an input memory image # slave_control –write slave1 < slave1_image
Create the slave CPUs memory image from the execution file # slave_control –create slave1 < slave1.exe > slave1_image

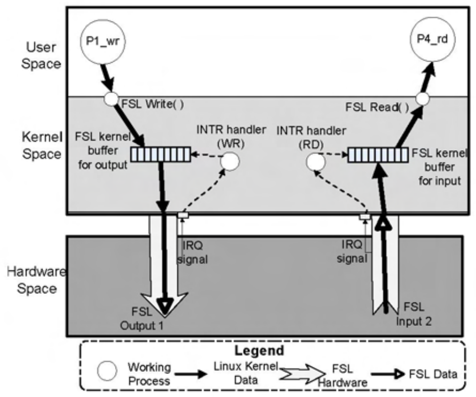

Fig. 2. Interrupt based FSL driver for Linux

control the Slave CPUs. These methods were implemented in a utility program (‘*slave_control’)* to perform the various actions. Table II describes the usage of this program.

B. FSL FIFO driver for Linux

FSL is the hardware channel for the master-to-slave and slave-to-slave communication. On the slave side, the blocking versions of the native MicroBlaze FSL instructions can be used, since there is only a single process of execution. However on the master side, such blocking read/write operation can cause the system to freeze or stutter by causing the entire OS, including interrupts, to be stalled until the FSL transaction is complete. Operating system design principles forbid direct access to hardware by application software, motivating the use of a device driver to mediate this access. Finally, a device driver can implement buffering which improves overall throughput by preventing applications from blocking when the relatively small hardware FIFO channels are full.

We kept the general design and structure of the original FSL FIFO driver [8], and extended it to operate in an interrupt-driven rather than polled mode. This gives much better performance when the master CPU is under heavy load and reduces overall kernel overhead.

Fig. 2 illustrates the simplified FSL driver internals including the read/write interface, kernel buffer and the interrupt handler. It only illustrates a single channel pair – in reality all 8 FSL channel pairs are managed by the same driver. With these changes the FSL driver is capable of sending/receiving multiple channels of FSL data with coherence and high performance under different user-kernel space system loads. The driver presents a standard Linux character device interface, supporting standard system calls such as *open(), read(), write() and close()*.

C. Ghost process and pipe redirection

A common multiprocessing methodology in Unix/Linux is to split an application into multiple independent processes, connected by software based FIFO channels, called pipes.

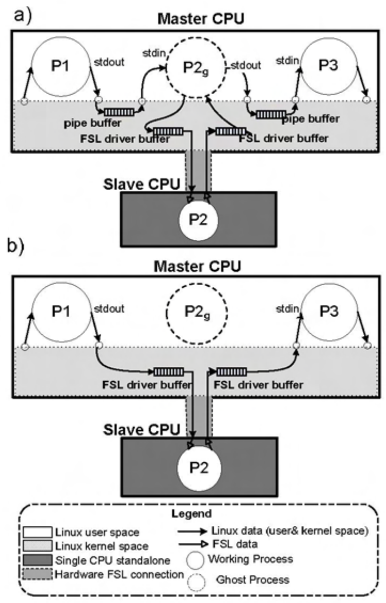

Fig. 3. Ghost process pipe redirection

Similarly, the slave processors can be connected to the master processor’s process through an agent process inside the Linux user space. This methodology can provide a process model that supports additional slave processors inside the same operating system, including support for commonly used Linux inter-process communication methodologies. We call this process model “*ghost process*”, and the details are described as follows.

For clarify and brevity, the following examples use conventional command line shell script syntax and constructs for process and pipe creation. However, the system programming calls such as *fork()*, *exec()* and *mkfifo()* could also be used in a controlling (master CPU) application to achieve the same effect. Consider three processes ***P1***, ***P2*** and ***P3*** connected by the pipes, executed by the following command line:

```
# P1 < input | P2 | P3 > output
```

In the event that the ***P2*** process workload must be migrated into the slave CPU, a ghost process called ***P2$_g$*** is created inside the user-space representing the agent for the slave CPU execution. The command line execution of the Ghost Process is similar to the previous command, where the difference is P2$_g$, the ghost process, replaces the original P2:

```
# P1 < input | P2g | P3 > output
```

Using this approach, ***P2$_g$*** will not do any real computation work, instead just transfer the data to and from the slave CPU’s ***P1*** through the FSL device driver. The slave CPU will execute the actual process ***P2***, and ***P1*** and ***P3*** are still entirely ignorant of the fact that ***P2*** is now being executed on a

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase® Smarter legal research.