FIFO Communication Models in Operating Systems for Reconfigurable Computing

J.A. Williams, N.W. Bergmann, X. Xie
School of ITEE, The University of Queensland
Brisbane, Australia
{jwilliams,n.bergmann, xxie}@itee.uq.edu.au

Abstract

Increasing demands upon embedded systems for higher level services like networking, user interfaces and file system management, are driving growth in fully-featured operating systems such as embedded Linux. In reconfigurable System-on-Chip (rSoC) design, a critical issue is efficient integration of custom hardware and software resources, where efficiency must be considered in terms of both design time and run time. Process networks communicating via FIFO queues are a powerful model for real time digital system design, especially for data streaming applications such as multimedia devices. FIFOs also form a central part of Unix and Linux Interprocess Communication (IPC) architectures, where they are more commonly known as pipes. In this paper, we expand on this observation and show how the combination of embedded Linux, reconfigurable System-on-Chip, and FIFO communication models provide a compelling platform for efficient design- and run-time implementation of complex, high performance embedded systems.

1. Introduction

One of the central challenges in hardware/software codesign is the decoupling of hardware and software processes. How can the various computational components of a system be specified and designed in such a way as to make transparent their instantiation either as custom hardware in logic gates, or sequential software executing on one or more microprocessors?

In this paper we consider FIFO-based interprocess communication (IPC) models, and show how they fit naturally into a reconfigurable operating system that treats hardware processes (computational functions implemented in hardware rather than software) as first class system objects.

In software operating systems like Linux, FIFOs (known more commonly as pipes) are a key IPC mechanism that supports loose coupling between separately developed software components. Similarly, FIFOs are commonly used as the communication mechanism in process network systems – a popular methodology for design and implementation of real-time systems.

The primary contribution of this work is the insight that by bringing together these two approaches, previously considered related only in a conceptual manner, we take a significant step towards one of the loftier goals in reconfigurable computing – the development of a unified architecture for heterogenous hardware/software system implementation.

2. Motivation

Our long term goal is to map hardware processes into the regular Linux process space, complete with logic allocation and dynamic hardware processes. Some fundamental capabilities have already been demonstrated, such as dynamically self-reconfiguring Linux systems running on soft-core processors [1]. The requirements of such a system are summarised below[2]:

1. **support sequential (processor-based) execution**, with a *familiar programming paradigm as a starting point for application development*.;
2. **offer interoperability with existing general purpose computing infrastructure**, including networking, file storage and other I/O device interfacing;
3. **provide a process model that seamlessly supports hardware, software, and hybrid processes** within the same architecture, including *support for standard interprocess communication methodologies*;
4. **provide a logic management interface** that abstracts operations such as dynamic partial reconfiguration, in support of the hardware process model;
5. **support integration of hardware components developed in a variety of tool flows**;
6. **be scaleable**, supporting single-chip, multi-chip and multi-board computing systems.

Here, we are considering requirement 3 - how the Linux IPC mechanisms described above can be extended such that hardware processes, as well as software processes, may be used as communication end points. This concept is illustrated in Figure 1 below. The natural affinity of FIFO models to the Linux operating system and their utility in real-time hardware/software system design make them a natural approach to consider first.

3. Background

Much has been written in support of process networks connected by FIFO queues for modelling and implementing real-time systems. Kahn's Process Networks [3] are one wll-known example. Process networks are a natural match to data-stream processing systems, commonly found in real-time streaming and data processing applications. The combination of high-level synthesis with process network computation models promises a design environment whereby real-time data processing systems may be specified once, perhaps in a C-like language, and then be automatically mapped into a reconfigurable computing device, with processes implemented as a heterogenous mixture of hardware and software.

It is becoming frequently more common that an embedded system requires functionality well outside the traditional, narrowly defined behaviour found in older embedded systems. This is particularly the case for consumer-oriented devices. In addition to the "core" functionality, it is more likely that an embedded device will be expected to support standard networking protocols such as TCP/IP, as well as application-level protocols such as embedded webservers for configuration and control These factors are driving the rise of embedded operating systems, most notably embedded Linux. As hardware and memory resources become cheaper and more powerful, the productivity gains from using a standardised and well-known platform begin to outweigh any memory and performance overheads that may result.

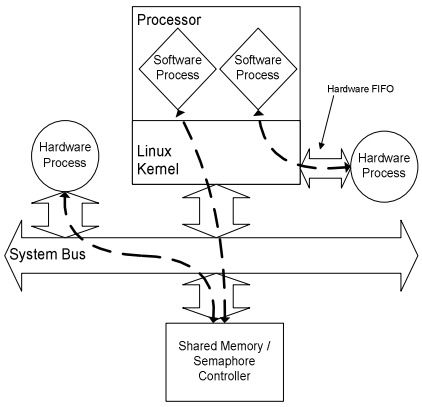

Figure 1. Hardware / software Interprocess Communication

Much of our recent research has looked at the applicability of embedded Linux in the newer domain of Reconfigurable System-on-Chip (e.g. [4]). The motivation for using a fully-featured, "conventional" operating system is quite simple. While certain aspects of reconfigurable computing are truly novel with respect to classic software systems, many of the same problems still apply. It is our contention that instead of throwing out these conventional methodologies – and with them the skills of thousands of system designers –it is better to integrate support for this new class of computational device into an existing context.

The main mechanisms available for inter-process communication are FIFOs or pipes, and shared memory/semaphores. FIFOs are essentially a special type of file that may only be read and written and opposing ends. Processes write data into one end, and read data from the other end, as though they are regular files, with the operating system managing the FIFO object in the kernel address space. Shared memory is typically implemented by mapping the same physical memory page into two (or more) processes' virtual memory spaces. Semaphores are provided to allow processes to synchronise their actions on this shared memory.

4. Approach

We are currently using the Xilinx Microblaze soft-processor for our reconfigurable Linux work. Of particular relevance here is the Fast Simplex Link (FSL) interface. FSL is a unidirectional, point-to-point bus interface, with a directed register mapped interface to the processor. Microblaze has eight each of FSL master and slave ports. FSL buses themselves are implemented as 32-bit wide FIFOs of parameterisable depth.

In an earlier paper, we showed how a network of programmable coprocessors could be connected to Microblaze via FSL channels, and how these channels could be mapped into the Linux environment as regular FIFO-like devices [5]. Upon reflection, we realised that this communication approach could be generalised – yielding a generic FSL FIFO driver that can allow any software process (Linux application) to communicate with a hardware process connected to the Microblaze via an FSL channel.

The FSL FIFO channel is implemented as a device driver, that maps to device nodes /dev/fslfifo0…7. In line with Unix philosophy, the fslfifo device is a regular Linux character device node, and implements kernel level IO buffering. This is in addition to the hardware buffering provided by the hardware FIFOs in each FSL channel. Communicating with a hardware process on the other end of an FSL connection is simple, the FIFO file is simply opened and then read or written to as required. From shell script, it can be as simple as

```
$ cat data.bin > /dev/fslfifo0
```

which will cause the specified data file to be streamed via the FSL FIFO device, off to the desired custom hardware. Similarly,

```
$ cat /dev/fslfifo0 > mydata
```

will stream data from the FSL-connected device. In this way, a hardware process attached to an FSL port may be treated like an input or output device. In the role of a computational accelerator, or coprocessor, a simple C program can open the fslfifo node in read/write mode, writing data in, then reading back the resulting processed information. The source code for the FSL FIFO device driver has been released under the GPL as part of the standard uClinux kernel source distribution – http://cvs.uclinux.org.

5. Conclusions and Future Work

We have described a simple yet effective approach to integrating custom hardware within embedded Linux. By mirroring the standard Linux pipe/FIFO IPC mechanism, it allows software processes to communicate with the custom hardware in a seamless and transparent manner.

Future work will include measuring and improving the performance of the FSL FIFO driver architecture – we have so far operated under the axiom that existence and useability are more important than raw performance. We are working on experiments with more sophisticated hardware process cores, with the goal of demonstrating both improved performance and improved design efficiency.

More broadly, the presented work is a step towards our overall goal of fully integrating custom hardware into the embedded Linux context. Our research so far indicates that this is a very promising line of enquiry, and we will continue to report our findings as they progress.

6. References

[1] J. A. Williams and N. W. Bergmann, "Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip," in Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA '04), Las Vegas, Nevada, 2004.

[2] J. A. Williams and N. W. Bergmann, "Reconfigurable Linux for Spaceflight Applications," in Proc. Military and Aerospace Programmable Logic Devices (MAPLD 04), Washington DC, USA, 2004.

[3] G. Kahn, "The semantics of a simple language for parallel programming," in Proc. IPIF '74, pp. 471-475, Amsterdam, 1974.

[4] N. W. Bergmann, J. A. Williams, and P. J. Waldeck, "A Flexible Platform for Real-Time Reconfigurable Systems on Chip," in Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms, pp. 300-303, Las Vegas, USA, 2003.

[5] J. A. Williams and N. W. Bergmann, "Programmable Parallel Coprocessor Architectures for Reconfigurable System-on-Chip," in Proc. IEEE Int. Conf. on Field Programmable Technology (FPT04), Brisbane, Australia, 2004.