Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip

John Williams and Neil Bergmann
School of ITEE, University of Queensland
Brisbane, Australia

Abstract - We have previously argued the benefits of embedded Linux as an operating system platform for reconfigurable system-on-chip design. In this paper we describe our approach building tools for the implementation of dynamically and self-reconfigurable systems, and show that embedded Linux is a natural and powerful platform on which to build these tools. We present examples and demonstrations that show how complex operations such as obtaining partial bit streams from remote servers and initiating reconfiguration are achieved with a single line of Linux shell script.

Keywords: dynamic self reconfiguration Microblaze embedded Linux

I. Introduction

The capability of modern SRAM-based FPGAs to be dynamically and partially reconfigured at runtime (a dynamically reconfigurable system, or DRS), without interrupting the operation of other logic within the FPGA, presents intriguing possibilities for novel system architectures and applications. This capability has been recognised and discussed at least since the advent of modern FPGAs if not before, however it is only recently that the technologies and tools have developed to the point whereby this may be considered a viable approach for practical digital systems.

The implementation of DRSs is exceptionally challenging. Previous practical successes have generally demonstrated one specific aspect or capability, at the cost of significant engineering effort. This disproportionate effort distracts from the real objective, to design and implement meaningful systems employing dynamic self reconfiguration.

Our approach to DRS design and implementation is to develop a platform of tools with which complex reconfigurable systems may be easily constructed. In this paper we propose embedded Linux as a natural host for such a platform.

As part of our reconfigurable system-on-chip (RSoC) research project called Egret [2], we have previously ported an embedded Linux kernel called uClinux, to the Xilinx Microblaze soft-core processor [3]. The capability to support research and experimentation into dynamic and self reconfiguring systems is one of Egret's design requirements.

To support this goal, we have integrated support for Xilinx FPGA self-reconfiguration into the Microblaze uClinux kernel, using the standard Linux device driver model. By leveraging the power and flexibility of the Linux platform, we are able to rapidly develop tools to perform complex dynamic self reconfiguration tasks.

The following section presents some brief background material on the Egret platform and the use of embedded Linux in RSoC (Reconfigurable System-on-Chip), and on existing approaches to DRS design and implementation. We then detail our approach to providing support for these systems within the context of the Linux device abstraction model. This is followed by examples that demonstrate the benefits of our approach, and finally we conclude and discuss some of the further challenges that remain for DRS research and design.

II. Background

A. Egret and uClinux

Egret is a modular platform for RSoC research, developed by our group. The first version of Egret targets Xilinx FPGAs, utilising the Microblaze softcore processor, however the Egret concept is not tied to one particular vendor.

Central to the Egret philosophy is that complex systems should be designed by assembling the required hardware modules, and specifying the module combination to a software tool that constructs the appropriate FPGA configuration, as

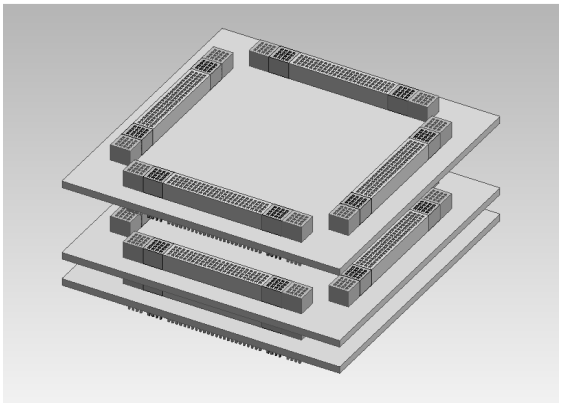

Figure 1. Egret module stack

well as software infrastructure such as device drivers etc. The Egret physical configuration is illustrated in Figure 1.

In line with this platform-based approach, it was determined that the software infrastructure needs of Egret would be best served by an embedded operating system, rather than a classical microkernel.

We chose the open source Linux derivative called uClinux ("*you-see-linux"*). uClinux is a port of the Linux kernel to support embedded processors lacking a memory management unit (MMU) [4]. From an application programming perspective, uClinux offers an interface almost identical to standard Linux, including command shells, C library support and Unix system calls. The uClinux kernel port to Microblaze was completed in 2003.

B. Dynamic and Self Reconfigured Systems

A three-axis classification scheme is useful to characterise the diversity of reconfigurable systems. The scheme classifies systems according to

- who controls reconfiguration,
- when the configuration is generated, and
- what is the level of reconfiguration granularity?

We discuss each of these below. These multiple axes represent continuous spectra of characteristics, rather than discrete points. The cases discussed below represent sample points along these axes.

1) Who Controls Reconfiguration?

We distinguish between systems whose reconfiguration is managed and controlled by some external device or host, and systems that initiate and control their own reconfiguration. Specifically

Exo-reconfigurable – from the Latin *exo-* meaning external. These are systems whose reconfiguration is initiated and controlled by an external source. The reconfiguration process is initiated externally. An FPGA coprocessor on a PCI bus is an example of this category.

Endo-reconfigurable – from the Latin *endo-* meaning inside, or internal. The decision to reconfigure, and the reconfiguration process itself are controlled autonomously by the system. A signal processing system that performs self-readback and small bit modifications to adjust filtering coefficients is one such system.

Hybrid – some combination of the above. A reconfigurable system that requests modules from a remote bitstream server falls into this category.

Until recently, most dynamically reconfigurable systems belonged in the first category, such as Xilinx's run-time reconfigurable crossbar switch [5], and the Cam-E-Leon project [6][1].

Xilinx recently reported connecting a Microblaze soft processor to the Internal Configuration Access Port (ICAP) of a Virtex2 FPGA [1], via the OPB microprocessor bus. This approach gives a Microblaze program access to the FPGA configuration system, to write configuration data and perform device readback etc.

2) When is the configuration data generated?

A dynamically reconfigurable system must load its new configuration data at runtime, but the question remains as to when that configuration data is generated. This is a spectrum ranging from fully static to fully dynamic configuration creation:

Static, Design-time – the loadable configuration data is determined fully at design time, including the relative placement of the reconfigurable modules within the device, and the connections of the modules to the rest of the system. All possible placements and variations of modules must be predicted in and synthesized in advance. This is the model supported by most vendor tools at the present time.

Run-time placement – pre-synthesised and routed hardware modules are dynamically modified to allow their placement at arbitrary locations on the reconfigurable device (e.g. [7]).

Fully dynamic module generation – configurations are generated dynamically according to run-time requirements. Modules might be synthesised from dynamically created VHDL or other hardware descriptions, or as instances created from a parametric module library.

---

[1] The Cam-E-Leon project also uses uClinux, but as the operating system running on a conventional embedded microprocessor device that manages FPGA configuration.

3) What level of reconfiguration (granularity)?

Systems can be characterized by the degree to which they manipulate the logic surface:

Small bit manipulations – the contents or configurations of individual FPGA logic elements such as LUTs are modified, but no overall logic or module-level changes are performed [7]. An example of this mode might be the dynamic modification of filter coefficients stored in FPGA LUTs.

Dynamically loaded modules – pre-implemented partial bitstreams are used to configure a portion of the FPGA's logic resources, to implement either a new functional module, or replace an existing module [7]. One can imagine a network encryption/acceleration co-processor, in which new hardware encryption modules might be swapped in at run time.

Many systems will fall somewhere between these two extremes. A reconfigurable system using internally pre-placed and routed modules that are dynamically modified to place at an arbitrary location is one example.

C. Summary

Several groups are developing systems and methodologies to manage the FPGA logic space, in terms of logic area assignment and partitioning (e.g. [8, 9]), and dynamic mapping of computations onto reconfigurable modules. This work will continue to have influence across the axes described above.

It is important to note that our research complements these existing approaches to DRS, and in fact enhances them by providing a high-level interface to the reconfiguration mechanism. It also permits direct translation of most exo-reconfigurable system concepts into endo-reconfigurable systems, by removing the requirement for the external controlling device.

III. Self reconfiguration in Linux

In this section we detail our approach to providing an abstraction layer for the Xilinx Internal Configuration Access Port (ICAP), and show some ways in which it can be used to implement dynamic self-reconfiguring systems. We first present a very brief introduction to the Microblaze system architecture and uClinux port, to provide context for the ensuing discussion.

A. Microblaze Architecture and uClinux Introduction

1) Microblaze

Microblaze is a classic 32 bit RISC processor, with 32 general purpose registers, and an orthogonal instruction set. It uses a 3 stage instruction pipeline, with delayed branch capability for improved instruction throughput.

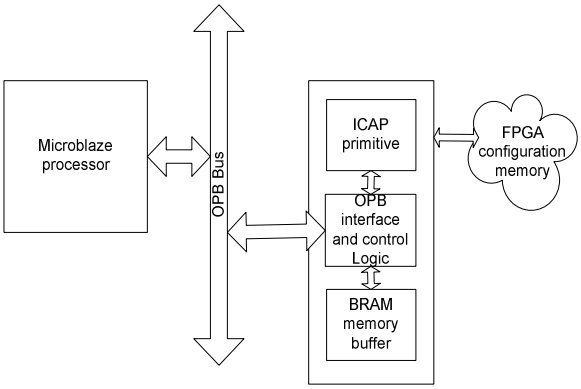

Figure 2. Architecture of the OPB-ICAP interface module (after [1])

The Microblaze design is specifically targeted to hardware features present in the various Xilinx FPGA devices, such as hardware multipliers and on chip block RAM (BRAM).

Microblaze utilizes Harvard-style separate instruction and data buses, which conform to IBM's CoreConnect OPB (On-Chip Peripheral Bus) standard. Bus arbiters can be automatically instantiated, permitting the instruction and data buses to be tied together, to create conventional von Neumann-style system architectures.

2) Microblaze uClinux

In most respects, the Microblaze port of uClinux is very similar to other ports to more conventional processors such as the Motorola Coldfire and ARM cores.

To minimize changes in the kernel memory architecture, the Microblaze systems used for uClinux are designed in the von Neumann style described above, with the instruction and data buses tied together. Peripherals such as timers, interrupt controllers, memory controllers, GPIOs and an Ethernet MAC are used to build up a complete system. Linux device drivers have been wrapped around these cores for interfacing with the kernel and user space applications.

B. The ICAP device driver

Xilinx developed an OPB interface to the ICAP module for self-reconfiguration experiments [1], which permits frame-by-frame readback and partial configuration in ICAP-supported devices. The OPB interface permits connection of this peripheral to the Microblaze soft-core processor. The architecture of OPB/ICAP interface is illustrated in Figure 2.

To integrate this device within the Linux kernel, we use the standard device driver architecture used by

all Linux devices. The Linux philosophy is that device drivers should implement mechanism, not policy, and this was adopted for the ICAP peripheral.

We developed a simple character-based device driver, which implements the read(), write() and ioctl() system calls:

read – initiates a read from the ICAP into a user memory buffer, of the specified number of bytes.

write – specified number of bytes are written to the ICAP from a user memory buffer

ioctl – interface to device specific control operations, such as querying the status, or changing operating modes

Upon system boot, this device is registered in the Linux device subsystem, appearing as /dev/icap. Like any Linux device, the ICAP may be accessed using standard Linux system calls, such as open, read, and write. Thus, the kernel mediates between user programs (implementing *policy*), and the device driver (implementing *mechanism*).

C. Using the ICAP device in a user program

Accessing the ICAP device from within a user program is simple:

1. `open()` the /dev/icap device
2. Construct a command sequence in a local buffer
3. `write()` the command sequence buffer
4. `read()` the result data (if applicable)

The format of the various command and data sequences is documented in the Xilinx Virtex2 User Guide [10]. For example, to perform a readback, a command sequence is constructed to

1. Issue dummy and synchronisation packets
2. Set the device ID code
3. Set the Frame Address Register (FAR)
4. Issue the ReadFrame command

After this command sequence is written to the device, the frame configuration data is read back as used as required.

IV. Linux shell programming for dynamic reconfiguration

One of the underlying principles of Un*x-like operating systems is to provide a collection of small tools, each focussed on performing a single job. The shell provides mechanism for chaining these tools together (e.g. pipes and output redirection). This approach makes the combination of uClinux and the ICAP device driver very powerful and easy to use.

The following sections present some examples of this approach. The intention is to demonstrate that the abstraction provided by embedded Linux and the ICAP driver allows one to focus on the interesting parts of the problem, rather than the detailed mechanics of the reconfiguration process.

A. Simple examples

A partial bitstream generated by the conventional logic synthesis and implementation tools is merely a sequence of configuration commands and data packets. Thus, given some partial bitstream present in a file system mounted on a Microblaze uClinux system, the reconfiguration process is performed simply by executing the command

```
$ cat partial.bit > /dev/icap
```

This elevates dynamic reconfiguration from being a low-level, complex procedure, to one which may be easily expressed and automated in much more accessible ways such as shell scripts.

It is worth noting that in this example (and those that follow), from an operating system perspective it makes absolutely no difference whether the file `partial.bit` exists in a local memory-based file system, an external disk file system, or even a remote network file system. Indeed, the first experiments in this work served the bitstreams over a Linux NFS (Network File System) mount from the development host machine.

From here, it is easy to see how complex dynamic systems can be constructed. We may develop a simple C program (or another shell script) that manipulates partial bitstreams, for example inserts or modifies coefficients in an FIR filtering module. A simple chaining together of commands performs the necessary operations:

```
$ cat filter_module.bit | set_coeffs
    0.1 0.4 0.4 0.1 > /dev/icap
```

B. Bitstream compression

The previous example demonstrates a form of bitstream compression. Partial bitstreams are highly structured, making it much more space efficient to store a generic ‘template’, along with information on how to specialise it (as per coefficient example above), than to create and store a large number of variations.

Standard Linux tools can also be used for bitstream compression:

```
$ gunzip –c bitstream.gzip > /dev/icap
```

```
#!/bin/sh
for i in “1 2 3 4 5”;
do
    wget –O /dev/fpga${i}
         http://www.bitstreams.com/
              bitfile${i}.bit
done;
```

Figure 3. Shell script to automatically retrieve remote bitstreams and configure FPGAs

Extending the filter coefficient example from before:

```
$ gunzip –c bitstream.gzip |
    set_coeffs 0.1 0.4 0.4 0.1 > /dev/icap
```

These examples show how complex operations may be performed by chaining multiple tools and utilities.

C. Networking and remote bitstream servers

We may easily leverage the other benefits of using a proper operating system – for example seamless integration of networking services. The “wget” command issues HTTP and FTP requests to remote servers. The following one-liner requests a bitstream (“partial.bit”) from a remote server (www.bitstreams.com for this example) and performs reconfiguration:

```
$ wget –O /dev/icap
     ftp://ftp.bitstreams.com/partial.bit
```

If our bitstream server is more intelligent, and can dynamically generate bitstreams according to some specified parameters, we can picture something like the following, to fetch an encryption module, dynamically specialised according to some parameters (the URL has been wrapped due to the short line length):

```
$ wget –O /dev/icap
    http://www.bitstreams.com/bitfile?
         mod_type=encrypt&param=…
```

Yet another possibility is that the bitstream server is implemented locally, such as would be the case for a purely endo-reconfigurable system. By utilising standard networking services, no change would be required to the client software:

```
$ wget –O /dev/icap
    http://localhost/bitfile?
        mod_type=encrypt&param=…
```

```
#!/bin/sh
# configure the driver hardware
cat conf_bus_driver.bit > /dev/icap

# load the kernel driver
insmod conf_bus_driver.o

# configure the slave FPGAs
for i in “1 2 3 4 5”;
do
    wget –O /dev/fpga${i}
         http://www.bitstreams.com/
              bitfile${i}.bit
done;
```

Figure 4. Shell script to dynamically load FPGA logic for configuring external FPGAs, and kernel module for device driver support

Simply substituting `localhost` as the server name is all that is required – the operating system takes care of the rest.

D. Self configuring FPGA arrays

We have so far considered the concept of a single FPGA and Microblaze system, managing its own reconfiguration. However, the concepts scale elegantly to the notion of FPGA arrays, with either a hierarchical configuration management strategy, or even a distributed/cooperative approach.

As previously mentioned, the ICAP is simply an internal interface to the FPGA configuration subsystem. The device driver approach presented here could just as easily be layered over an external configuration bus, used to configure arrays of FPGAs.

By exploiting the Linux device driver concept of major and minor device numbers, specific FPGAs in such a system could be assigned a particular minor number. The minor number would be used to control a chip-select signal on the configuration bus, and the configuration data streamed as appropriate. The shell script in Figure 3 would initiate the reconfiguration of an array of five FPGAs, connected to this “master” FPGA. In this example the bit streams are served remotely.

Note that the logic resources used by the master to configure other FPGAs are not required all of the time, so one might consider the approach of first swapping in the configuration bus driver hardware, followed by dynamically loading the device driver, as in Figure 4.

This shows how our approach permits thinking about the problem at a much more abstract level.

E. Readback and configuration verification

A small C program based on the readback algorithm presented in Section III.C was written. This is a very simple tool, taking as parameters the block, major and minor frame numbers (a

configuration frame address), and producing as output an ASCII representation of the configuration memory.

Such a tool makes it very easy to write programs and scripts to read back and verify the contents of the FPGA. The Xilinx implementation tool *bitgen* can produce as output a mask file that indicates which bits in a bitstream should be verified in a readback. Thus, by converting this mask file into the appropriate ASCII format, a readback and verification can be achieved with a simple readback followed by a comparison.

Action taken as a result of a successful or unsuccessful readback is application specific, but would likely involve reconfiguration using mechanisms like those described above.

Building on the idea of the previous section, using the same logic interface to the configuration of external FPGAs, a master device can very easily perform readback and verification of other FPGAs in a system.

V. Discussion

In the following we present discussion on some relevant aspects of the proposed approach that have not been previously addressed.

A. Performance

A performance price is always paid for the useability gained by higher level abstractions. Indeed, by choosing uClinux as the platform for our RSoC research, we accept the performance cost in exchange for the tremendous leverage offered by such a comprehensive platform.

In terms of the ICAP device and driver, the performance overhead is modest. Bitstream data is generated by an application (either dynamically, read from a file, or from a network connection). It is then sent to the kernel via the write() system call, which requires it to be copied once from user space to kernel space.

After being received by the kernel, the data is then copied into the OPB ICAP device's local memory. Finally, when the reconfiguration process is initiated, this hardware interface transmits the data to the actual ICAP core.

B. Application to other devices and systems

Clearly the ICAP resource in certain Xilinx FPGAs facilitates this research, however the concepts presented here can be applied to other devices and systems that do not have such built-in support for self-reconfiguration.

At the board level, general unconstrained user I/O pins cannot be routed around to the configuration pins of the device, and appropriate interfacing logic developed to provide the same capabilities as the Xilinx ICAP device and OPB bus wrapper interface mentioned in this work.

At that point, the software abstraction takes over, and a consistent, platform independent interface may be offered. Of course, the bitstreams themselves will be different from device to device, and that remains a challenge for all researchers in the reconfigurable systems domain.

C. Security and Bitstream Integrity

Working at the hardware level, ensuring the integrity of partial bitstreams is very important. Traditional design tools perform design rule checks (DRC) to prevent physical damage to devices from bad configurations, however they can do little to ensure that a bitstream will perform the intended function.

When a system is reconfiguring itself with bitstreams perhaps downloaded over a network connection, authentication and encryption become important. We argue that the Linux platform presents a natural solution. There already exist open source libraries that implement these functionalities, and it is a relatively simple matter to include them in the configuration sequence.

The more difficult problem is bitstream verification – how to determine (perhaps at runtime) that a given bitstream will not corrupt the current system operation. At a gross level, partial bitstreams can be inspected to determine their spatial range of influence. This information combined with a logic allocation map for the main system can be used to reject bitstreams that desire to make changes where they shouldn't. This checking could be added either at the user level or device driver/kernel level.

VI. Challenges and future work

So far we have deliberately avoided discussion on the mechanics of actually generating partial bitstreams, design modularisation, dynamic bitstream parameterisation and so on.

The practical difficulties facing researchers and practitioners in this regard are substantial. At the present time, synthesis and implementation tool support for these efforts is limited.

To implement and test our examples above, we used partial bit streams laboriously hand-created using the Xilinx FPGA Editor tool. This is partly because the modular and partial reconfiguration implementation flows are not supported for Microblaze and EDK (Embedded Development Kit) projects.

We have recently successfully "modularised" the Microblaze flow, and are in the process of automating this, so that reconfigurable modules

may be easily specified, and interfaced to Microblaze processor systems. This will greatly simplify the process of creating pre-defined modular bitstreams.

The issues mentioned previously in the discussion remain as important and fruitful avenues for further investigation.

VII. Conclusions

We have implemented and described a methodology and set of tools for implementing dynamically and self-reconfigurable systems, using embedded Linux as a powerful and flexible platform.

As the logic density and speed of FPGAs continues to increase, the relative cost of placing soft (or hard) processor logic in these devices diminishes. Similarly, the relative cost of using a complete embedded operating system such as uClinux also decreases. By adopting a platform based approach, designers and researchers can gain tremendous leverage.

By adopting the standard Linux device driver approach and philosophy, the ICAP reconfiguration mechanism becomes available to user programs, as well as higher level shell scripts. Examples were presented to show how complex behaviours such as remote network-based bitstream acquisition and reconfiguration could be implemented in as little as a single line of shell script code. Readback and configuration verification was shown to easily integrate within this framework.

The idea of using the same driver interface and architecture to control the configuration of arrays of FPGAs was proposed as a natural and simple extension of the approach.

One of the major challenges in the design and implementation of dynamic and self reconfiguring systems is to coerce the logic implementation tools to produce the appropriate partial bit streams, and also the dynamic modification of those bit streams to allow dynamic logic placement and other capabilities. There are a number of research and commercial groups working on these problems, and success in these areas could be readily translated into our platform and tool approach.

Acknowledgements

The authors would like to thank the members of the Xilinx EDK team and Xilinx Labs. This research is partly supported by the Australian Government via the Australian Research Council.

References

[1] B. Blodget, S. McMillan, and P. Lysaght, "A lightweight approach for embedded reconfiguration of FPGAs," in Proc. IEEE Design Automation and Test in Europe, pp. 339-340, Munich, Germany, 2003.

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

[3] Xilinx, "Microblaze Processor Reference Guide," Xilinx, Inc, 2003, pp. 136.

[4] A. Rubini and J. Corbet, Linux Device Drivers, 2nd ed: O'Reilly and Associates, 2001.

[5] G. Brebner and D. Levi, "Networking on Chip with Platform FPGAs," in Proc. IEEE International Conference on Field-Programmable Technology (FPT 03), pp. 13-20, Tokyo, Japan, 2003.

[6] S. Guccione, E. Verkest, and I. Bolsens, "Design technology for networked reconfigurable FPGA platforms," in Proc. Design, Automation and Test in Europe Conference and Exhibition, pp. 994-997, 2002.

[7] C. Patterson, "A Dynamic Module Server for Embedded Platform FPGAs," in Proc. ERSA, pp. 31-40, Las Vegas, USA, 2003.

[8] Y. Nakane, K. Nagami, T. Shiozawa, and A. Nagoya, "Concept and Implementation of Run-time Resource Management System Operating on Autonomously Reconfigurable Architecture," in Proc. IEEE International Conference on Field Programmable Technologies (FPT 03), pp. 136-143, Tokyo, Japan, 2003.

[9] G. Wigley and D. Kearney, "The Development of an Operating System for Reconfigurable Computing," in Proc. IEEE Symposium on FCCM, 2001.

[10] Xilinx, "Configuration Details," in Virtex2 Platform FPGA User Guide. San Jose, CA, 2004, pp. 293-306.