Operating System Support for Reconfigurable Processors in a Workstation Enviornment

Michael Dales

Department of Computing Science, University of Glasgow,

17 Lilybank Gardens, Glasgow, G12 8RZ, Scotland.

michael@dcs.gla.ac.uk

Abstract

Reconfigurable processors, which combine run time programmable hardware and microprocessors on the same chip, are becoming popular in the embedded systems domain. They allow application specific logic to be loaded onto the processor and then controlled from software. Such an ability could be useful for accelerating applications on a workstation. However, integrating such a device into a workstation environment is not a straight forward task. The reconfigurable hardware resource on such a device would need to be shared easily, fairly, and securely, which will require changes at the hardware, operating system, and programming environment level. In this paper we outline the problems that need to be overcome when managing this new resource, and how we plan to solve them.

1 Motivation

Field Programmable Logic (FPL) devices, such as Field Programmable Gate Arrays (FPGAs), are an increasingly attractive solution for providing custom hardware in embedded systems. FPL devices are Integrated Circuits (ICs) that can have their functionality defined at run time. An FPL device consists of a two dimensional array of cells, which typically contain a set of logic elements such as registers, multiplexors, and small Look Up Tables (LUTs). Each cell is surrounded by a set of wiring resources to which cells may be attached. By configuring the contents of each cell and which cells are attached to which wires, sequential logic circuits can, in effect, be loaded onto the device. The behaviour of the cells and routing is controlled by SRAM on the device. Configuration data, referred to as a *bitstream*, can be loaded into the SRAM to set the behaviour of the device; reconfiguring the device is just a matter of reprogramming the SRAM. Despite relatively slow configuration times (of the order of several milliseconds), the benefits of custom hardware are numerous, most notably it is cheaper for prototyping than generating custom ASICs, and it can be upgraded in the field.

A recent development is the coupling of microprocessors and FPL on a single device for use in embedded systems. Despite the benefits of custom hardware, a lot of problems still need software as part of the solution, typically for doing control flow. There have been both numerous commercial [1, 15] and research projects [7, 9, 12, 13] (to mention but a few) to address this need. In such a system, application specific logic, such as custom I/O interfaces or algorithm accelerators, can be loaded onto the FPL part of the device and then controlled from software running on the processor. Despite uptake in the embedded systems environment, there has been little work done on integrating such a device into a workstation environment. Moving from an embedded systems environment to a workstation environment, however, imposes a new set of demands and restrictions. Such integration will require changes to the application programming model, the operating system, and the hardware.

The first obvious difference between the two environments is that the finite FPL resource will need to be shared between a dynamic set of competing processes with unpredictable resource demands. Given the large number of processes a workstation can potentially run at a single time, and the scarcity of the FPL resource, it is vital that the operating system can share out the FPL resource fairly. Although modern FPL devices can theoretically support circuits of millions of gates, they are quickly consumed by a number of reasonably sized circuits, especially when being shared between applications. As a result, in addition to allocating FPL space, we expect the operating system to have to move circuits on and off the processor as demand dictates. Given that the cost of moving configuration data on and off an FPL device is quite high, this could potentially mean that the benefit of having the resource could be negated by the management overheads associated with it. Security in a workstation environment is also of great concern. Unlike an embedded system, where the system runs a limited trusted set of tasks, a workstation is much more open to malicious or accidental abuse. If a reconfigurable processor was misconfigured, then not only could the attack damage data, but it could also destroy the processor itself [5].

The focus of this paper is to outline how we plan to tackle the operating system management issues in integrating a reconfigurable processor into a workstation environment, based on previous work which demonstrated a suitable hardware architecture [4].

2 Overall Approach

The aim of this work is to examine how a traditionally structured operating system could be used in conjunction with a suitably structured reconfigurable processor (see Section 3). The idea is to demonstrate that a full workstation class operating system and programming environment could integrate the additional FPL environment. In this section we outline the basic approach we will take.

2.1 Application Interface

In our system, programs will come with a set of *custom instructions* that will be used to augment the existing processor instruction set with instructions tailored specifically for that process. The custom instructions will have the same interface as a traditional instruction on the processor, and may consist of either combinatorial or sequential logic. Custom instructions will be invoked using a machine code instruction that encodes an identifier, the Circuit ID (CID), specifying which circuit loaded on the FPL the process is attempting to invoke, and the operands specifying the source and destination locations. The only other functions a process requires are calls to the operating system to register and unregister custom instructions along with an associated CID. All other operations, including the placement and loading of custom instructions, should be handled by the operating system.

2.2 Operating System Tasks

The operating system is charged with managing allocation of the FPL resource between the set of active processes. In a modern operating system, processes are presented with a virtual machine in which they appear to have sole access to a resource. It is the operating system's job to multiplex all the processes' use of the virtual resource onto the physical resource. A good example of this is virtual memory. Processes have a large private address space, parts of which the operating system will map onto physical memory as needed. Physical memory may contain parts of many virtual address spaces at once. The processes refer to objects in their virtual address space using virtual addresses, which must be translated to physical addresses before being used to access physical memory. This is all handled in the operating system and hardware, so the process is never aware of the virtualisation.

We propose a similar model for the FPL resource. Processes will have a virtual instruction space, which can contain as many instructions as there are possible CIDs. When a process attempts to execute a custom instruction, the operating system will attempt to load the custom instruction as needed. If there is no room on the FPL then it will move existing circuits off the processor using an eviction policy to make room for the incoming circuit. Once the circuit has been loaded then it will need to program the processor to make a mapping between the process's CID and the appropriate block of FPL.

2.3 Rich Custom Instructions

Previous work in harnessing reconfigurable processors has treated the configuration bitstream as the custom instruction, but we propose a much richer data structure. We split the configuration data into two parts: that pertaining to stateful elements in the circuit (the registers), and the remainder which describes the stateless elements (the LUTs, routing, etc.). This division provides several benefits. Firstly, it significantly reduces the amount of data that needs to be preserved when a circuit is evicted from the processor (by 91% in our prototype architecture described in Section 3). Secondly, it allows the operating system to potentially share circuits. If two or more processes are trying to use a common configuration, then the operating system can simply replace the stateful elements in the circuit, rather than having to swap the entire circuit.

Although it is the aim that careful scheduling of custom instructions by the operating system will mean that configuration overheads do not slow down the system so as to negate the benefit of the resource, this may not always be possible. To help the operating system cope when the FPL resource becomes heavily oversubscribed, processes should provide a software alternative along with the configuration bitstream. The operating system can then use this to ensure that processes make progress despite not being granted part of the FPL resource.

Thus a custom instruction becomes a combination of the two bitstreams, the software alternative and associated meta data (e.g., status information, symbolic name). Rather than simply being a concatenation of these parts into a single object, a custom instruction is defined as a structure of references to the separate parts. This makes sharing and dynamic linking of parts possible [10].

3 Basic Processor Architecture

In previous work we have proposed the Proteus Architecture [4], which is an outline for a reconfigurable processor with the FPL resource designed to be managed by an operating system. The Proteus Architecture places the FPL inside an additional function unit on the processor, similar to a traditional integer or floating point unit. The new function unit contains a register file and a set of FPL blocks of equal fixed sized called Programmable Function Units (PFUs), into which instruction definitions can be loaded. The PFUs' functionality is specified at run time by loading configuration bitstreams, allowing the processor's instruction set to be tailored for the currently running set of processes. Configuration of FPL takes a relatively long time, so the Proteus Architecture first loads the configuration into an on-chip FIFO, which can then be drained in parallel with the processor executing other software.

To enable the operating system to virtualise the reconfigurable resource, processes do not directly invoke specific PFUs. The operating system will be responsible for loading custom instructions into PFUs, so a process does not know at link time where the instruction will be loaded. The custom instruction may be loaded in a PFU, may be being dispatched to software, or may not be loaded yet. To support this, a dispatch mechanism is added to the processor to decode CIDs as they are invoked. The dispatch mechanism will either convert the CID to a PFU reference and decode the instruction as a PFU invocation, convert the CID to a memory address at which the software alternative is located and decode the instruction as a branch instruction, or the CID will not be decoded and the operating system will be invoked to handle the fault. It is the operating system's role to ensure that the dispatch table has a correct set of mappings to ensure system integrity and security. To prevent the dispatch hardware from requiring to be flushed on a context switch, each mapping contains the PID of the process it belongs to. This is used to form a system unique mapping for each instruction.

To aid the operating system in making good circuit eviction selections, the processor provides two basic counters with each PFU. One counter tracks the number of times that the associated PFU has been invoked since the register was last reset, the other counter contains the value copied from a global counter which is increased every time a PFU is either reconfigured or invoked. This provides sufficient information for basic eviction policies such as LRU, second chance, LFU, and so on.

We have built a simulation model of a prototype architecture, based around the ARM processor, which adds a reconfigurable execution unit to the ARM core. The reconfigurable unit consists of a 32 by 32 bit register file and eight PFUs, which are based on the Xilinx Virtex FPGA fabric [14]. The PFUs are of sufficient size to hold a reasonably complex circuit. For example, we have an alpha blend instruction that sums two 32 bit RGBA pixels, which requires ten Booth's multipliers, three dividers, and three adders. This architecture has been demonstrated to provide a group of basic programs running individually on the system with a substantial performance increase.

4 Operating System Structure

For this work we are assuming an operating system structure similar to that of a Unix implementation, such as BSD, or a Microsoft Windows NT derivative. This means that we have a pre-emptively multitasked, multiuser environment, where each process has a private virtual address space.

The focus is to extend the operating system with a Custom Instruction Scheduler (CIS), which is responsible for managing processes' custom instructions. Processes will make a system call to associate a custom instruction with a CID. It is then the operating system's responsibility to ensure the custom instruction executes when the process requests it. There are obvious parallels between managing PFUs and virtual memory pages.

The basic model uses a demand-loading system, whereby when a process first invokes a CID, the dispatch mechanism fails and causes the operating system to be invoked, at which point the custom instruction can be loaded. If there is a free PFU then the operating system can load the custom instruction's bitstreams into it, load the appropriate mapping into the dispatch hardware, and then reissue the faulted process. If there are no free PFUs, then the operating system will have to make a policy decision and either choose an instruction to evict from a PFU, or use the software dispatch. In earlier work we have noted that configuration costs for a single circuit can be amortised over a reasonable scheduling quanta (we tested as low as 1 ms), so the default policy would be to evict a custom instruction, using on a history-based algorithm similar to those used for page eviction. The operating system will need to preserve the state of the evicted circuit if necessary before reloading the PFU, updating the dispatch mapping, and reissuing the process. During loading the processor can run other code, allowing the operating system to issue another process to execute whilst the original process is blocked during loading.

Although we believe that for light levels of contention the basic scheduler outline above will suffice, it is possible for the FPL resource to become congested due to excessive requests. This problem is particularly noticeable if a process uses multiple instructions in a tight loop, where it might have problems getting all the instructions loaded at once, and thus spend most of its time waiting for circuits to load. To solve this, we plan to investigate the use of a second level scheduler, based on run-time statistics monitoring similar to that used to calculate process priorities in BSD [8]. Using a periodic monitor, the operating system will attempt to detect when PFUs are being reconfigured excessively. At this point, the operating system can switch policy and start moving custom instructions to their software alternative, ensuring that all processes can again make progress.

In addition to scheduling circuits, the operating system is in a position to attempt to manage how they are used, in particular trying to share circuit usage if at all possible. It is possible that common libraries of custom instructions may be used (e.g., a SIMD arithmetic library), and the operating system should attempt to map multiple circuit instances onto the same PFU, reducing a complete circuit switch to just a potential state switch. To do this the operating system needs a way of knowing when custom instructions are the same. Note that we are only concerned whether the static bitstream of the circuit is the same; there is no benefit from sharing the rest of the custom instruction. Doing a bit-wise comparison of each instruction as it is registered is potentially expensive. Instead, we provide each instruction with a short name, such as an MD5 checksum, which can be compared during registration; a match would then cause a bit-wise comparison to be used. All this would be done transparently so the process does not need to manage sharing.

5 Related Work

Although there has been no work to date examining the management of a reconfigurable processor in a workstation environment, there has been a lot of research in managing FPGAs. For systems where a FPGA is connected to a controlling processor, there have been attempts to provide middleware that manages the FPGA, similar to our CIS. Both [2] and [3] define an interface whereby processes register circuits with the system for later use, then request that the system loads their circuits, and finally request their invocation. The scheduling in this system is made more complicated due to the problems of scheduling arbitrary sized circuits on an FPGA.

Another related avenue of research is that of configuration caching, which considers when to load circuits onto an FPGA for use, in an attempt to minimise the number of configurations required [6, 11]. Techniques are proposed for predicting FPL usage: off-line analysis, invocation patterns, and recent history techniques. However, the work is based on the more predictable embedded systems environment, and only the recent history techniques would be useful in the dynamic and unpredictable workstation environment.

6 Conclusion

We have outlined the need for investigation into operating system support for reconfigurable processors for use in a general purpose workstation environment. We have outlined a possible way of solving the problems, and are currently attempting to implement a small operating system kernel with CIS to test the practicability of this technique.

The author would like to thank his proof readers, Prof. Joe Sventek and Jonathan Paisley, for their invaluable input, and to Xilinx who have sponsored the work.

References

[1] Altera. *ARM Based Embeeded Processor Device Overview*. Altera, 2001.

[2] Gordon Brebner. A virtual hardware operating system for the Xilinx XC6200. In *6th International Workshop on Field Programmable Logic and Applications*, pages 327–336, September 1996.

[3] J. Burns, A. Donlin, J. Hogg, S. Singh, and M de Wit. A dynamic reconfiguration run-time system. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 66–75, April 1997.

[4] Michael Dales. Managing a Reconfigurable Processor in a General Purpose Workstation Environment. In *Design, Automation, and Testing in Europe*, March 2003. to be published.

[5] Ilija Hadžiǵ, Sanjay Udani, and Jonathan M. Smith. FPGA Viruses. In *9th International Workshop on Field Programmable Logic and Applications*, pages 291–300, September 1999.

[6] Scott Hauck, Zhiyuan Li, and Katherine Compton. Configuration Caching Techniques for FPGA. In *IEEE Workshop on FPGAs for Custom Computing Machines*, April 2000.

[7] John R. Hauser and John Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 12–21, April 1997.

[8] Marshal Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quaterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley Publishing Company, 1996.

[9] Rahul Razdan and Michael D. Smith. High-Performance Microarchitectures with Hardware–Programmable Functional Units. In *Proc. 27th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 172–180, November 1994.

[10] J. H. Saltzer. Naming and Binding of Objects. In *Number 60 in Lecture Notes in Computing Science*, pages 99–208. Springer-Verlag, 1978.

[11] Sejar Sudhir, Suman Nath, and Seth Copen Goldstein. Configuration Caching and Swapping. In *11th International Conference on Field Programmable Logic and Applications*, pages 192–202, August 2001.

[12] M. J. Wirthlin and B. L. Hutchings. A dynamic instruction set computer. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 99–107, April 1995.

[13] Ralph D. Wittig and Paul Chow. OneChip: An FPGA Processor With Reconfigurable Logic. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 126–135, 1996.

[14] Xilinx. *The Programmable Logic Data Book 1999*. Xilinx, 1999.

[15] Xilinx. *Virtex-II Pro Platform FPGA Handbook*. Xilinx, 2002.