Managing a Reconfigurable Processor in a General Purpose Workstation Environment

Michael Dales

Department of Computing Science, University of Glasgow,

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

michael@dcs.gla.ac.uk

Abstract

Reconfigurable processor hybrids are becoming an accepted solution in the embedded systems domain, but have yet to gain acceptance in the general purpose workstation domain. One problem with current solutions is their lack of support for the dynamic workloads and resource demands of a general purpose workstation. In this paper we describe and demonstrate a reconfigurable processor architecture that lets the operating system dynamically share the FPL resource between a set of applications without the management overheads negating the benefit of having the extra resource.

1 Introduction

There has been recent research and commercial interest in bringing together Field Programmable Logic (FPL) devices and microprocessors in a single device. Although FPL devices, such as Field Programmable Gate Arrays (FPGAs) provide a flexible medium in which to produce problem solutions, some activities, like general control flow, are better handled in software. However existing solutions assume an embedded systems approach, where the FPL is the main focus. Instead we are interested in letting software applications on a general purpose workstation benefit from access to reconfigurable logic in order to accelerate their core algorithms. This requires a different focus to designing such a processor, particularly with respect to the interface between the software and hardware. In a general purpose workstation the number of applications wanting to use the FPL and the amount of FPL they will require cannot be predicted and will be constantly changing. As such we need an effective way of sharing the FPL resource dynamically, fairly, and securely, which is the focus of this work. We describe a suitable interface between the software and custom hardware on a reconfigurable processor such that an operating system can manage the FPL and share it out between competing requests from other applications.

The paper is structured as follows: in Section 2 we discuss our system requirements, and in Section 3 relate this to existing work. In Section 4 we propose and describe a new architecture which fully supports the requirements outlined in Section 2. In Section 5 we demonstrate that the architecture is practicable, and in Section 6 we conclude the paper.

2 System Requirements

We wish to produce a system where software applications running on a general purpose workstation using a traditional operating system can take advantage of FPL to speed up their core algorithms. The system must support a dynamic mixture of traditional pure software applications and applications accelerated with custom hardware. The demands on the FPL resource made by applications will be dynamic and unpredictable, and may at times exceed the physical limits of the resource. In this system the operating system should be able to manage these dynamic requests and share out the resource between applications, ensuring all applications make timely progress.

We envisage a simple interface between an application and its custom hardware. The custom hardware cores will behave like new instructions within the system, allowing applications to tailor the instruction set to their own needs. Applications will register custom instructions with the operating system using a process unique Circuit ID (CID); the operating system will then be responsible for ensuring that the new instructions can be found when they are needed by the application, without the application having to explicitly load instructions onto the FPL. To help during times of contention, as well as including a hardware description, an application may provide a software alternative to the instruction. The operating system can defer execution to the software alternative rather than swapping circuits on and off the processor if the FPL is full. We are interested in seeing if

this arrangement will have a benefit for scheduling custom instructions.

Multiple applications may wish to use multiple circuits at once; given the size of modern FPL devices, we feel it should be possible for multiple circuits to reside on the array at once, to reduce the need for moving circuits. However, sharing the FPL can cause the system to suffer from internal and external contention. Internal contention is when an application is forced to have only a single circuit on the FPL fabric at once. This could lead to excessive swapping if an algorithm requires more than one circuit in a tight loop. External contention occurs when multiple applications are using circuits simultaneously, but the array can only support a single application's circuits on the array at once; if more than one application is using custom instructions then applications will need to reload their circuits after each context switch.

In addition to new management problems, adding an FPL resource to a processor in a general purpose workstation raises a new set of security issues. Physically, it makes it possible for applications to physically damage the processor and attached devices through misconfiguration [4]. Functionally, the operating system needs to ensure that circuits will behave correctly, responding to events such as interrupts and terminating in a timely fashion.

3 Related Work

The notion of processor hybrids is not new, with there being several existing research projects and commercial products based along these lines.

Commercial solutions such as the Xilinx Virtex–II Pro [13], Altera Excalibur [1], and Triscend A7 [10] offer one or more microprocessor cores on the same die as an FPL resource. These devices connect the processor and the configurable logic using a memory mapped interface. Custom hardware cores are connected to a bus containing address, data, and control lines which are connected to the processor's memory bus; the processor may then access the devices as if they were memories. The cores are responsible for responding to a particular range of memory addresses, which are set at design time. This interface works well for embedded systems, but is less suited to a more dynamic environment. Although the memory mapped solution, given a sufficiently large segment of the address space, prevents internal contention, this solution suffers from external contention. It is not reasonably possible to ensure that no two applications will use the same address ranges for their circuits. A possible solution is to allow the operating system to set the address ranges used in the circuits before they are loaded, and then program the virtual memory map for the application appropriately, though this requires the operating system to modify the applications' bitstreams. Another drawback with these devices is that the application programmer has to carefully manage the memory interface on the processor when communicating with custom hardware to achieve best performance. The MMU and caches expect to be talking to memory devices and optimize transactions on this assumption. This places more strain on the programmer and adds to the latency of using custom hardware. Finally, traveling off the processor and across buses to custom hardware is itself quite slow compared to traditional data processing operations. Ideally our solution would not need to move off the standard processor datapath during computation.

Combining FPL into the datapath of a processor has been the basis for many research projects, such as PRISC [7], CoMPARE [8], GARP [5], the SHARK DSP Hybrid [3], and OneChip [11]. Combining the FPL resource into the processor's datapath greatly simplifies the interface between the two parts, with custom hardware being accessed directly by using special instructions. This simplifies the interface and reduces the issue latency, but at the cost of a less flexible interface and less bandwidth. The bandwidth issue can be tackled partly by using a wider register file to feed the FPL, similar to how modern processors use wider register files to feed SIMD units [6], or by modifying the datapath to allow more than the conventional two registers to drive the FPL, as is done in CoMPARE. The main concern with the existing approaches are the restrictions in sharing the FPL resource. Architectures such as CoMPARE, GARP, and the SHARK DSP Hybrid allow only a single circuit to be loaded onto the array at once, an approach which suffers from both internal and external contention. OneChip allows multiple circuits to reside on the FPL array, but only as part of a single configuration, so it reduces, but does not remove, internal contention and still suffers from external contention.

The most flexible approach is that taken by PRISC, which is aimed at a workstation like environment. PRISC uses a set of Programmable Function Units (PFUs) into which applications can load combinatorial circuits that can then be called using a traditional instruction call. Each PFU has an ID register into which an application specific opcode for that circuit is loaded. When an instruction invokes a PFU, the opcode in the instruction is compared with the registers: if there is a match the circuit is used, otherwise a processor exception is thrown allowing the operating system to respond to the event. The separate PFUs solve the problem of internal contention, and by wiping the ID registers on a context switch and reloading them with a particular process's IDs as it needs them, it also solves the problem of external contention. There are drawbacks with the PRISC architecture however. Firstly, we would like custom instructions to use sequential logic to allow for more interesting applications. Secondly the dispatch mechanism

is not very flexible: it does not support multiple opcodes for circuits, meaning circuits can not be shared internally, and requires the ID registers to be reset on a context switch. Despite these drawbacks the PRISC architecture is the best approach of those discussed for a workstation environment.

4 The Proteus Architecture

Although there are a lot of good architectures combining reconfigurable logic and a microprocessor, no one architecture is completely suitable for our target domain, so we propose a new architecture that meets all the requirements for supporting reconfigurable logic in a workstation like environment. Our approach described below is similar in general layout to the PRISC architecture, but differs in terms of the dispatch mechanism and uses a richer FPL fabric, which dramatically changes the run–time management costs.

We propose placing the FPL resource into a new function unit on the processor to sit alongside the traditional units, such as integer and floating point units. This unit will contain its own register file and a series of PFUs connected to that register file with a traditional two word input/one word output interface as used for other instructions. Applications access their custom instructions by using the CID they have associated with them. When the instruction is decoded, the dispatch mechanism will convert this CID into a physical PFU reference.

4.1 The FPL Fabric

Although the focus of this work is the management of the circuits on the array, the makeup of the fabric has important consequences on the management costs and system security. First, similar to the PRISC architecture, we do not require I/O Blocks (IOBs), as the PFUs only connect to the processor datapath, and do not need to interface directly with device pins. This removes one potential security threat, as having IOBs would give potential for software to cause physical damage to the system by driving pins incorrectly [4]. In addition we assume a mux based routing fabric, which prevents the array from being misconfigured such that short circuits can occur. In our initial ProteanARM implementation (see Section 5) we assume an array based on the Xilinx Virtex fabric [12], which uses mux based routing.

Unlike PRISC, we want a stateful FPL array in the PFUs to allow more complex sequential logic based circuits. This has two important consequences on the architecture: state must be preserved as circuits move in and out of PFUs and a mechanism is required to ensure that instructions in PFUs terminate (which we address in Section 4.4). Although we all some state, we believe that application state should reside in either the register file or main memory, so FPL state should be kept to a minimum, which means we allow registers in CLBs, but not the large RAM blocks found in modern FPGA fabrics.

Moving configuration data on and off the processor at times of contention adds a significant overhead to the system, reducing the benefit of having the new resource; for example, in the ProteanARM each custom instruction requires 54 Kbytes of data to be transfered for a configuration. We can reduce the amount of data that needs to be transferred by noting that we do not need to save entire configuration, just the configuration information for the stateful elements. Thus we split the configuration into two sections: configuration for static elements, like LUT contents and routing, and configuration for loading state into CLB registers. The hardware should either support two separate configuration places, or allow a partial configuration containing just the state information to be stored and loaded separately form the rest of the configuration.

4.2 The Dispatch Mechanism

The dispatch mechanism is responsible for mapping an application's request for a circuit using the associated CID to the appropriate custom instruction, i.e., dispatching the custom hardware or the nominated software alternative, or if no suitable mapping occurs notifying the operating system. The PRISC system, with its ID registers for each PFU, provides an adequate mechanism but has some limitations we would like to remove: it needs reprogramming on every process switch, does not support mapping multiple opcodes to a single circuit, and does not support the software alternative mapping.

We want a system that will take a PFU execution instruction in the decode stage of the processor pipeline and resolve it in one of three ways. The preferred resolution is to match the current process's CID to a PFU. If a match is found then the instruction is decoded as a invocation of a custom hardware core in a PFU. The next option is that a mapping has been recorded between the CID and the memory address of a software alternative. If this mapping is found then the instruction is decoded as a special branch to the software alternative (see below for more details on the software dispatch). Finally, if no mapping is found then the processor will cause an exception to occur, causing the operating system to be invoked. The operating system can then either terminate the process if the mapping request was illegal, or load the custom instruction and reissue the application from where it faulted.

The dispatch mechanism needs a way to uniquely map a process's CID to a given custom instruction instance. To create a globally unique namespace we combine an application's CID for a custom instruction with the Process ID (PID) which is already held on the processor in workstation

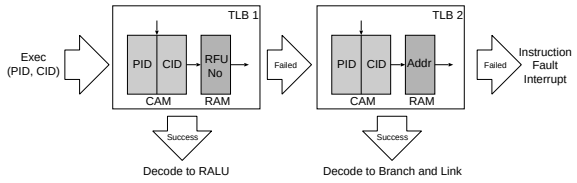

Figure 1. The proteus dispatch mechanism.

class processors. The PID and CID pair provide a system unique ID tuple for referring to a custom instruction, which means the mapping hardware will not need to be flushed on a context switch like PRISC. An important distinction to note is that an ID tuple is not the absolute name of a custom instruction, but rather a custom instruction can have many ID tuples associated with it to facilitate sharing custom instructions. This complicates the mapping hardware; it is no longer sufficient to just associate an ID tuple with a PFU. Instead we use a Translation Lookaside Buffer (TLB) arrangement, as shown in Figure 1. A TLB consists of a Content Addressable Memory (CAM) used to store ID tuples which is used as an index into a RAM containing either the PFU number in which the circuit resides for circuit dispatch, or the software address for software dispatch. This arrangement allows for the mapping of multiple ID tuples to a single circuit or software routine. This has one drawback: more mappings may be needed that can fit in the TLB, so a custom instruction that is loaded in hardware may fault if its mapping has been pushed out the TLB. When the operating system sees a custom instruction fault it must first check if it is just a mapping fault before attempting to load the hardware.

4.3 Software Dispatch

Dispatching to a software alternative for custom hardware requires the destination function to be able to decode the original instruction to work out which operands it has to use. This can be a time consuming operation, so we provide hardware support to speed this up.

The solution we have chosen is to make the FPL unit remember the operands in special purpose registers, big enough to hold the two source operands and the result operand. These registers are filled during a special branch instruction used to move execution to the software alternative. Special load/store instructions can then be used that work with the special purpose registers to provide the data access needed without the routine even needing to see the original operands explicitly. Additional instructions also exist to allow the operating system to read and write the registers directly, allowing them to be preserved over a process switch. One subtle problem with this mechanism is that if the software alternative uses a custom instruction which also dispatches to software then the contents of the special purpose registers will be lost, potentially before they were finished with. However, we consider this bad practice: using a custom instruction whilst trying to reduce contention on the PFUs does not make sense; execution only reached the software alternative since there was not room for custom instructions in hardware.

4.4 Long Instruction Handling

In the Proteus Architecture custom hardware instructions may run for multiple cycles. Long running instructions have important consequences on a processor though, in that they either must have an upper bound on the number of cycles they can take to ensure they do not lock up the processor or increase interrupt latency, or the instructions must be interruptible. The simplest solution is to simply limit the number of cycles a custom instruction can take, but we would prefer for instructions not to be constrained in that fashion, so have opted to make the system interruptible. During execution of an instruction it can be interrupted by processor exceptions and then restarted from the point at which it was interrupted transparently without the application being aware that this occurred.

The interface to PFUs is designed with two additional control signals: an init signal going in and a completion signal coming out. When a custom instruction is first invoked the init signal goes high for a cycle, indicating to the custom hardware that this is the first cycle of an invocation, allowing it to set itself up accordingly. The circuit is then clocked until the completion goes high, which tells the processor to stop clocking the PFU and to store the value produced by the PFU appropriately. When an instruction is interrupted, we can continue execution simply by reissuing the invocation instruction and not setting the init signal to high. We do this by using a 1 bit status register to feed back the completion signal into the init signal. On reset all the status registers are set to 1, so when an instruction is started the circuit will see the init signal high. For subsequent cycles the done signal will flow through the register and set the init signal low. If the instruction is then interrupted and reissued the init signal will be low on reissue, so execution will continue as if nothing had happened. On completion the done signal will go high, placing a 1 in the status register again ready for the next invocation.

4.5 Usage Statistics

To aid the operating system in deciding which circuits it is best to swap off the array during periods of contention, the set of PFUs each have associated with them a register containing a count of the times that instruction has completed. These registers can be read and cleared by the op-

erating system. It can then use these registers to implement classic scheduling algorithms such as Least Recently Used (LRU), Second Chance, etc. [9]. Note that we take the count at the end of the instruction rather than at the start to allow for instructions to be interrupted and reissued.

5 Demonstration

The ProteanARM is a demonstration of the Proteus Architecture. It originally appeared in [2] without management support, to demonstrate that the basic datapath was sufficient to provide applications with a performance increase. We extended the simulator model to include the modifications outlined in Section 4.

The ProteanARM is based on an ARM7TDMI processor, and adds the reconfigurable execution unit to the datapath as an on-chip coprocessor, the standard way of adding additional function units to the ARM. The coprocessor consists of a 16 element 32 bit wide register file connected to a set of PFUs and the dispatch hardware described above. For experimental purposes we limited the processor to four PFUs of 500 Configuration Logic Blocks (CLBs) (we estimate that the chip could support twice that number of PFUs, but limit it in order to demonstrate the system behaviour under contention). The only change we had to make to the ARM core was to change the coprocessor interface and control logic to allow the coprocessor to generate a memory address for the software dispatch mechanism, which is not possible under the normal interface. Otherwise all the changes are limited to the coprocessor unit.

POrSCHE (Proteus Operating System and Configurable Hardware Environment) is a simple operating system kernel developed from scratch to demonstrate the ProteanARM platform is practicable. It uses a simple pre–emptive round robin process scheduler to run multiple processes. The basic POrSCHE kernel without PFU support runs successfully on actual ARM hardware. POrSCHE implements a Custom Instruction Scheduler (CIS) as part of the kernel, which manages the circuits registered with the OS by different applications. The CIS is responsible for loading and unloading circuits and for managing the dispatch hardware.

5.1 Experiments

To demonstrate the platform working we ran two initial scheduling experiments, testing both basic scheduling and using software dispatch as an alternative to circuit swapping. For each experiment we ran three sets of runs with between 1 and 8 instances of a particular test application: alpha blending image processing, twofish encryption, and audio echo processing. Two of the test applications (alpha blending and twofish encryption) use a single custom instruction so should cause contention after four processes

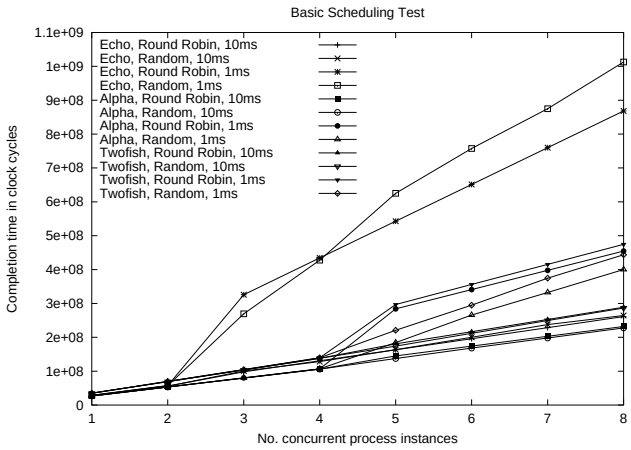

Figure 2. Results for basic scheduling test.

are run concurrently, and the other uses two custom instructions in a tight loop, so contention should occur after just two concurrent instances. Each test uses two scheduling quanta sizes to attempt to show the difference between batch scheduling and interactive scheduling. A scheduling quanta of 10ms was used to demonstrate batch scheduling based on the Linux batch scheduling quanta. A shorter quanta of 1ms was used to indicate performance on a more interactive system where applications are not getting their full quanta. In the final system applications using the same circuits would attempt to share instances, just changing the state in a single PFU; however we are interested in the effect of overloading here, so sharing is not allowed.

5.1.1 Circuit Switching Test

In this test we ran each complete set with a round robin and random circuit replacement policy. These results can be seen in Figure 2. In all cases the increase in completion time is linear with the number of concurrent processes until PFU contention occurs. This occurs after four processes for the test applications with a single circuit and after two processes for the test application with two circuits. At this point context switch overheads reduce the overall performance. At a 10ms quantum value the extra overhead has only a small effect of completion times, however at the 1ms quantum value the increased number of switches causes a more significant performance reduction. The round robin policy generally performs worse than the random policy in most cases. This is due to bad interaction with the round robin process scheduler, which typically means applications lose their circuits after a context switch. Note that all runs performed an order of magnitude faster than the unaccelerated applications.

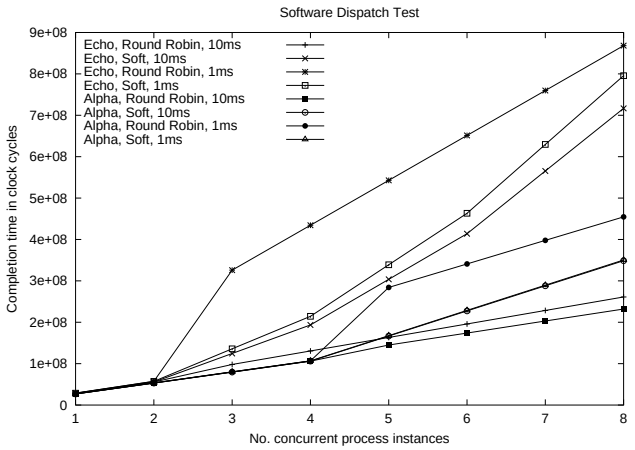

Figure 3. Results for software dispatch test.

5.1.2 Software Dispatch Test

These results for the alpha blending and audio encryption tests can be seen in Figure 3 (the twofish example follows a similar trend to the alpha blending results and is not shown). The graph shows the same points of contention as before, but the subsequent performance of the software dispatch implementation differs significantly. The change in scheduling quanta has little impact on performance of the software runs due to the lack of circuit switches. The performance of software dispatch runs lies between the 10ms and 1ms runs with circuit switching. This indicates that the software dispatch routine is only useful when an application suffers many circuit switches.

5.1.3 Discussion

The above experiments demonstrate that we can successfully manage the FPL resource on the processor using the management facilities we have provided, without losing the performance benefit of having the additional resource. The cost of moving circuits on and off the array proved sufficiently small that software dispatch had no advantage for the runs using 10ms scheduling period. In other operating systems, such as Windows NT and BSD variants which use a batch scheduler period of 100ms, the benefits would be even better. The software dispatch mechanism proved useful only during periods when applications just get short quanta, such as might be found in an interactive system. However, this ignores the added delay during configuration that a virtual memory system would add (circuit may need to be paged in from disk), and as such software dispatch may yet prove an interesting option.

6 Conclusion

With this work we have demonstrated a reconfigurable processor architecture that is suitable for use in a workstation environment, where the operating system manages the FPL resource and the management does not negate the benefits of having the FPL.

Having done the basic work we are now going on to consider the higher level issues about how a programmer utilizes such a system from a programming language perspective and to test the performance of the system with more dynamic scheduling loads.

References

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

[2] M. Dales. Initial Analysis of the Proteus Architecture. In *11th International Conference on Field Programmable Logic and Applications*, pages 623–627, August 2001.

[3] P. Graham and B. Nelson. Reconfigurable Processors for High–Performance, Embedded Digital Signal Processing. In *9th International Workshop on Field Programmable Logic and Applications*, pages 1–10, September 1999.

[4] I. Hadžić, S. Udani, and J. M. Smith. FPGA Viruses. In *9th International Workshop on Field Programmable Logic and Applications*, pages 291–300, September 1999.

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

[6] Motorola. *MPC7400 RISC Microprocessor Technical Summary*. Motorola Semiconductor Products Sector, rev 0 edition, August 1999.

[7] R. Razdan and M. 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.

[8] S. Sawitzki, A. Gratz, and R. G. Spallek. CoMPARE: A Simple Reconfigurable Processor Architecture Exploiting Instruction Level Parallelism. In *Proceedings of the 5th Australasian Conference on Parallel and Real-Time Systems*, pages 213–224, 1998.

[9] A. Silberschatz and P. B. Galvin. *Operating System Concepts*. Addison–Wesley, 4th edition, 1994.

[10] Triscend. *Triscend A7 Configurable System–on–Chip Family*. Triscend Corporation, 2000.

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

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

[13] Xilinx. *Virtex–II Pro Platform FPGA Handbook*. Xilinx, 2002.