A Virtual Hardware Operating System for the Xilinx XC6200

Gordon Brebner

Department of Computer Science
University of Edinburgh
Mayfield Road
Edinburgh EH9 3JZ
Scotland

Abstract. Computer operating systems relieve a user of the responsibility for managing physical resources, such as processors, memory and input/output devices. The evolution of FPGA technology means that a new resource is available — it is accessed like memory, but can behave like a flexible processor or input/output device. There is a rôle for an operating system in making this resource easy to use, by managing it on behalf of a user. This paper explores the issues involved in such management, in particular the problems involved in presenting a software-oriented user with a larger virtual hardware resource that is implemented using smaller physical FPGA hardware. A prototype operating system, that demonstrates operational solutions to the problems using the Xilinx XC6200 FPGA, is described.

1 Introduction

This paper explores ways in which the flexibility and rapid reconfigurability of FPGA hardware can be harnessed in an effective way by conventional processor-memory subsystems. This is a fundamental question that will affect the future of FPGA technology, and aspects of it have been discussed in other work, notably by Hutchings and others [4,8]. The issues raised here straddle the fuzzy line between hardware and software within the hardware-software co-design of systems. The overall intention is to make the FPGA hardware available as an extra resource to operating systems. In doing so, the FPGA hardware can be viewed variously as having the nature of processors, memory or input/output devices, or some combination of these. This gives a range of possible models for virtual hardware sub-systems.

The performance of FPGA technology is always going to be poorer than that of tailored ASIC technology. Thus, benefits must come from flexibility and dynamic re-use — in short, savings in hardware real estate, while still providing significant speed benefits over purely software solutions. If FPGA technology is harnessed in a general-purpose way using an operating system, then extreme care is necessary to ensure that the hardware and software overheads of the operating system do not swamp any benefits gained. This fact must be a guiding light whenever postulating any operating system functions.

The work reported in this paper involves the new Xilinx XC6200 FPGA technology [5]. Two novel features of this FPGA family are of particular interest. First, cells can be reconfigured rapidly and selectively using write operations at SRAM speeds. Second, addressible registers can be configured on the cell array, to allow interaction with other system components. Together, these features allow an FPGA to be treated physically as a memory device, and logically as a memory-mapped processing or input/output device, as well as being a hardware component interacting using signals transmitted through input/output pins. This allows experimentation at the hardware-software boundary.

2 XC6200 tools

Conventional circuit design tools are provided with the XC6200 family: high-level language-based or schematic logic design, followed by placement and routing, and finally the generation of configuration data for the cell array. This allows the rapid implementation of hardware designs using the XC6200 technology. Such a provision is analogous to supplying high-level language compilers for software. The machine code generated by most compilers is not complete, but instead is intended to run with operating system support. In turn, the operating system is responsible for making efficient use of resources on behalf of the high-level language program, and so must contain machine-specific components. We feel that a similar approach is wise for virtual hardware. Rather than having total translators that produce optimised monolithic FPGA configurations from structured high-level descriptions, we favour tools that translate high-level designs to rather more abstracted configuration data that makes use of operating system facilities. Specifically, use of these facilities is expressed by the fact that configurations conform to patterns supported by the operating system.

The operating system is essentially concerned with managing the overall configuration of an XC6200 cell array. To support operating system design experiments, a collection of tools has been constructed to manipulate configuration data expressed in its lowest form: the binary data written to cells. A data structure describing the configuration of logic functions, flip-flops and interconnections in an array is central to the tools. Two tools can populate this data structure:

1. a library of C functions that can be used to describe the configuration of cells, flyover wires and input/output pads — essentially, an 'assembly language' level of programming; and
2. an input function to read standard format `.cal` files containing configuration data, generated by the XC6200 design tools or other sources.

Two tools can make use of the contents of this data structure:

1. an instrumented simulator to allow the behavioural simulation of the configured array using inputs from its configured input/output pins; and

2. an output function to write standard format `.cal` files containing configuration data corresponding to the data structure contents.

The simulator allows all of the features of the XC6200 architecture to be experimented with, in particular, different means of interaction with other system components. Limited real experimentation is possible using an XC6200 evaluation board connected to a PC via a serial link; in the future, better and realistic experimentation will be possible with the availability of a PCI bus XC6200 board for the PC.

Further details of these tools, and their usage, is not a subject of this paper. The C library for low-level circuit design is described in [1], and a thorough account of the design and use of the simulator can be found in [6]. The main contribution of the tools to the work described here has been to allow rapid experimentation with operating system ideas.

3 Virtual hardware models

In some respects, the underlying model for virtual hardware is similar to that for virtual memory. In both cases, an illusion of much greater hardware resources is created for the user. This is achieved by sharing of the actual physical resource over time. A key property of any such scheme is that sufficient physical resource must be available so that the costs of swopping material into and out of the physical resource can be amortised over a period of time between swops during which meaningful amounts of work may be carried out.

The fundamental difference between virtual hardware and virtual memory is that, in general, virtual hardware is considerably more structured. Virtual memory can be seen as a collection of independent memory locations and, in principle at least, swopping can be done at the grain of a single memory location. In practice, of course, swopping is done at a larger grain, for example, physical pages or logical segments, which acknowledges the fact that there is usually some structure in the way that memory locations are accessed over time. This improves efficiency, but is not an essential.

In the case of logic circuits, however, a collection of components operate as an inter-connected whole. It is not possible to swop components independently, because the circuit will no longer work. Timing and usage of components is an integral part of the circuit, and so it is far less easy for an operating system to observe, and decide when components are not in use and so eligible for swopping out.

Thus, although the FPGA is physically interfaced as an additional memory device, its computational properties preclude any naïve extension of a virtual memory model to encompass it. We define a **swappable logic unit** (SLU) to be a logic circuit capable of performing a function in terms of specified input(s), and then supplying results on specified output(s). Two key features of an SLU are that it has a fixed area in its FPGA implementation, and that its input and output interfaces are fixed as part of its design. Of course, a single memory location is an example that fits within this definition.

There are various possible models for input and output arrangements that can be made for an SLU:

- signals on the perimeter of the SLU;
- bus-accessible registers within the SLU; or
- active bus access by the SLU.

These models are familiar from the sphere of interfacing input/output devices to processor-memory subsystems, although the SLUs are more akin to processing elements in nature. However, there are two important novel features. First, perimeter signals can be used to interact with other SLUs. This opens up the issue of how larger circuits might be decomposed into a collection of swoppable SLUs. Second, SLUs compute at hardware rates, and so interaction with other components via the input and output arrangements is not subject to the significant delays associated with input/output devices.

The guiding principle of any model should be simplicity, so that it is possible for a virtual hardware system to be implemented without significant run-time overheads. A further merit of simplicity is to minimise the number of constraints on the design of circuits that form SLUs. A crucial issue is how to find SLUs that are small enough to be sensible units of swopping, while being large enough to perform useful computational functions.

There are two other issues which are related to making swapping efficient. One is to exploit similarities between different SLUs, so that one SLU can be swapped for another using partial reconfiguration of the differing FPGA cells only. A particular case of this is where parameterised SLUs are designed, which can be instantiated with particular cells in their parameterised areas. The second issue is to exploit repeated patterns of cells within an SLU, so that an SLU can be swopped in more quickly by parallel configuration of the repeated parts. Both of these matters rely intimately on the actual FPGA implementation of SLUs, and illustrate a matter in which an operating system must be tailored to the available hardware in order to improve its operational efficiency.

Motivated by the above discussion, we have identified two different virtual hardware models that have scope for having fairly wide applicability, without being unduly expensive when implemented. Each fills a different computational niche, and allows useful FPGA properties to be harnessed. The first model is a **sea of accelerators** model, in which the SLUs are independent units capable of performing computational functions, made available via bus-accessible registers. The second is a **parallel harness** model, in which the SLUs are cooperating parallel processing elements, interacting with neighbours using perimeter signals and wiring infrastructure supplied by the operating system. Note that, for such a parallel system, if there is an irregular interconnection pattern and relatively infrequent communication, then the first model, with a processor acting as a sequential communication device between SLUs, might be a better solution.

For both models, there is no attempt to make an operating system analyse arbitrary circuits to determine suitable decomposition into SLUs at run time. Any such analysis would be too expensive. Thus, the virtual hardware systems are not being presented as general-purpose schemes for accommodating large circuits using smaller shared real estate. They are being presented, respectively, as a way of managing a large collection of small circuits and as a way of managing a circuit composed of a collection of regularly-connected small circuits. We feel that any less structured possibilities are not practicable without fatally hampering efficiency. Our experimental programme is intended to investigate whether even these restricted models are practicable and, if not, how FPGA technology might be enhanced to make them practicable.

Note that these models are intended to be independent of specific FPGA technologies. Thus, the technology-independent parallel harness model should not be confused with the issue of identifying replicated patterns within SLUs. The latter is technology-dependent, and is targeted at optimising the implementation of the operating system for a specific technology. Another way of looking at this is to say that the parallel harness model is an abstraction of parallelism at execution time, whereas SLU pattern identification is to aid parallelism at configuration time.

Assuming the practibility of implementing these models, then the next stage is to develop compilation tools that take high-level descriptions of systems, and map them to a collection of SLUs that fit within one or other model. For example, a high-level language compiler could map simple library functions on to SLUs that are swapped using the sea of accelerators model. Alternatively, a compiler for a language like Occam might map replicated functions executed in parallel on to SLUs that are swapped using the parallel harness model. The intention is that the operating system makes available a more software-like model than would be available directly from the FPGA technology.

4 Implementation using the XC6200 FPGA

As remarked earlier, two important features of the XC6200 architecture are partial configuration and configuration of bus-accessible registers. There are two extra useful features associated with these. First, a wildcard facility is available to assist with partial configuration: the same value may be written into several or many cells simultaneously. This gives scope for speeding up configuration when many similar or identical elements are being used within SLUs. Second, registers need not be arranged as a collection of adjacent flip-flops. Map registers can be used to specify that a register is formed from bit slices located in separated positions in one column of the array. This gives scope for flexibility in designing circuits that must contain interfacing registers. Both of these features are examples of technology-specific features that an operating system can manage as a service to a user.

A guiding principle of the implementations is that SLUs should be position independent — that is, they can be placed anywhere on the cell array that is convenient for the operating system. Two features of the XC6200 technology mitigate against this. First, flyover wires of length 4, 16 and 64 can only begin on 4-cell, 16-cell and 64-cell boundaries respectively. We allow this as a feature of SLUs that restricts position independence. Second, input/output pads may be present, and these can only be placed on the perimeter of the array. In the current implementation, we disallow such pads, in the spirit that SLUs are 'user' resources, and only 'system' resources should be allowed access to the real input/output pins. We foresee a need to allow some SLUs to carry out real input/output, so that input/output processing elements or active bus accessing units can be added in the future, but do not expect this to pose any major technical problems.

The prototype system contains support for only the sea of accelerators model. However, through implementing support for replicated patterns within SLUs and parameterisation within SLUs, the essential mechanisms for supporting the parallel harness model have been developed. A simple two-dimensional grid harness can be simulated by an SLU containing a grid of replicated elements, each element having superimposed parameterisation. Interaction between elements is possible by local SLU wiring between abutting elements. This style of interconnection within FPGAs has been used in other work, for example [3] and [7]. In addition to this simulation, the arrangements needed for a tree style harness have already been investigated in earlier work [1]. This harness was used in an interesting case where each SLU in the harness represented one state of a state machine, with the harness representing possible transitions. The fact that only one SLU is computationally active at a time leads to good opportunities for swapping of individual SLUs while the overall circuit is in operation.

For the sea of accelerators model, each different SLU is represented by a descriptor, which includes information about its size and the position of any input and output registers. There is also a pointer to the configuration data required for the SLU. In addition to this data, there is additional information to make swapping more efficient. There is an optional list of parameterised areas, to allow partial reconfigurations that adjust the behaviour of the SLU without completely reconfiguring it. There is also optional pattern replication information, to specify the size of replicated elements and the number of replications present. An $m \times n$ replication of an element causes mn copies of this element to be placed as abutting neighbours in an area of size mn times the element size. The operating system does not police the interconnections between replicated elements. This is the responsibility of the SLU circuit designer, or the design tools used.

The main responsibilities of the operating system are for placement of required SLUs within the cell array, for ensuring that any registers in SLUs are bus-addressible, and for allowing user access to the appropriate bus addresses. The services of the operating system are made available to a user via a library of C functions. Full details of the implemenation of the prototype operating system and the C library can be found in [6].

The first main function allows a user to declare the existence of an SLU. The information supplied as parameters is used to construct a descriptor for the SLU, and this descriptor is added to a list maintained by the operating system. Each SLU only needs to be declared once, and this must be before it is instantiated on the FPGA.

The second main function allows a user to instantiate an SLU on the actual FPGA. This facility enables an SLU to be swapped in 'by hand', although this should not be necessary in a final operating system, where the operating system is responsible for automatically swopping in SLUs as their computational abilities are required. If the SLU has parameterised areas, function parameters are used to specify the actual logic that should appear in each such area.

In implementing this function, the operating system first has to find sufficient free space on the cell array to accommodate the SLU. A two-dimensional recursive bisection technique is used. The smallest free area that is big enough is chosen. Then, this area is successively halved in size along its longer dimension until an area just big enough for the SLU is obtained. This algorithm tends to maximise the number of square holes that are left available as free space. Such holes are more likely to be usable than narrow rectangular holes. If there is no big enough free space for the SLU, the operating system removes least recently used SLUs until enough space is available. In addition to this housekeeping on demand, there is periodic tidying of the whole cell array, to remove holes that have appeared in the array due to the release of SLUs. Such housekeeping is possible as long as SLUs are designed to be largely position independent.

After allocating enough space, the SLU can be configured into the cell array. This is done in a way that makes use of any efficiencies possible through parameterisation and replication. If there is re-parameterisation of an SLU already present, then only partial reconfiguration of the relevant areas is necessary. If there is replication of elements within the SLU, then the wildcard facility is used to configure multiple parts of the SLU in parallel.

The final act of the operating system is to store map register values that reflect the positioning of the SLU's input and output registers, given the position chosen for the SLU on the cell array. These values, together with the array column numbers in which the registers have been configured, allow input/output to be performed via the system bus.

The third main function allows a user to access the computational capabilities of an SLU. The parameters to the function are an identifier of an SLU and input values for its registers. The function returns the output values from the SLU's registers. Given that the SLU operates at hardware speeds, a stop-and-wait approach is used between supplying the inputs and collecting the outputs. Thus, the overall appearance presented to a user is of an extra high-speed library of functions being made available, or of an extended instruction set being made available, depending on the level of programming being used.

The current operating system acts merely as an intermediary in passing inputs from a user to the SLU, and return outputs from the SLU to the user. An enhancement being implemented involves marking certain input or output registers as implementing a standard control interface between the operating system and the SLU. One use of this would be for circuitry to request configuration changes to itself. If the circuit enters a state that requires partial reconfiguration to take place, it will be able to signal this to the operating system via status bits in a control output register. Another use would be for the operating system to supply an input timing signal for circuitry with sequential behaviour that is clocked at the input rate determined by the operating system or a user.

5 Preliminary results

Preliminary experiments have been carried out with a simple collection of SLUs. Each SLU is based on a bit-slice unit to perform a simple function, with this unit being replicated different numbers of times. For example, one SLU family is based on a one-bit full adder, with family members for 8-bit, 16-bit and 32-bit addition. The fact that each SLU is formed using a replicated element leads to substantial gains through using the wildcarding facility of the XC6200. This means that the configuration time for each member of the family is the same, that is, a 32-bit adder can be configured as quickly as an 8-bit adder. More importantly, the configuration time is comparable with the time for required for only a few input-compute-output cycles that use the circuitry.

Assuming a 40 ns access speed to the FPGA from the system bus, an 8-bit or a 32-bit adder can be fully configured in 720 ns (without wildcarding, the 8-bit adder would take 5760 ns, the 32-bit adder four times more). The time for one 8-bit addition is around 300 ns, and the time for one 32-bit addition is around 600 ns.

Similar results were obtained for the other bit-slice SLUs. Of course, this experience has to be extrapolated with extreme caution. If parallel configuration through wildcarding is not possible, then SLU swopping times will be much increased. General efficiency will come from ensuring that SLUs are reasonably small, but are sufficiently useful that they can stay resident for sufficient uses to amortise the cost of swopping.

Aside from timing measurements, the experimental testbed used so far has demonstrated that the prototype operating system can successfully manage a large collection of varying-size SLUs, allowing a program to make use of each SLU as an accelerator for some computational function. The FPGA real estate is managed invisibly by the operating system, with the user program unaware of where circuitry is located on the cell array.

6 Conclusions and future work

The work conducted so far gives grounds for hope that a virtual hardware model can be implemented efficiently. This will not be as wide-ranging and general-purpose as the well-established virtual memory model, but will allow two key architectural classes — accelerators and homogeneous parallel processors — to be handled. If an attempt was made at more general types of circuit decomposition, it seems that an asynchronous FPGA technology would be required, to allow independent operation of circuit components and the temporary absence of components.

Future work, already under way, will involve experimental investigations using practical applications. This involves, first, looking for ways to map the computational needs of particular applications onto the two models (by hand, in the first instance), and then measuring the speed trade-offs between pure hardware implementation, pure software implementation, and a virtual hardware implementation. These experiments will also allow comparison of virtual hardware solutions with other FPGA-based solutions, in particular, those that exploit rapid reconfigurability. One further particular area of interest, that will require extensions to the operating system, is to investigate circuitry with registers and computation units of parameterised size, that can grow or shrink to reflect the physical resource available.

The two models can be seen as a technology-dependent bridge between software and hardware. Future compilation tools will be developed to compile software to an intermediate form that makes use of operating system functions. This form would refer to SLUs in terms of their input-output specifications. Individual SLUs would be designed using FPGA circuit design tools targeted at specific technologies. The operating system then manages this collection of SLUs and the FPGA real estate in order to implement the operating system functions required by software.

The Xilinx XC6200 FPGA architecture has various features that make it attractive as a vehicle for the implementation of virtual hardware. However, some of these features are provided in a way that appears to be more for the convenience of the chip designer than the virtual hardware operating system implementor. We expect that this work will suggest improvements for a new generation of the architecture, much as traditional computer architecture evolved to include instructions and facilities to expedite virtual memory systems. One example is the wildcarding feature, which allows a certain amount of parallel configuration, but could be much improved to accommodate repliacted patterns that appear in real circuits. An improved form of this feature has already been investigated by incorporating it into the XC6200 simulator.

Acknowledgement

Many thanks are due to Damian Kwok for lengthy discussions that have helped to shape the ideas in this paper, and also for his implementation work on both the XC6200 simulator and the prototype operating system.

References

1. Brebner: "The SPODE circuit design library for the XC6200," Internal report, Department of Computer Science, University of Edinburgh, 1996.
2. Brebner and Gray: "Use of reconfigurability in variable-length code detection at video rates," Proc. 5th International Workshop on Field Programmable Logic and Applications, Springer-Verlag 1995.
3. Eldredge and Hutchings: "RRANN: a hardware implementation of the back-propagation algorithm using reconfigurable FPGAs," Proc. IEEE International Conference on Neural Networks, 1994.

4. Hadley and Hutchings: "Design methodologies for partially reconfigured systems," Proc. IEEE Workshop on FPGAs for Custom Computing Machines, 1995.
5. Kean, Churcher and Wilkie: "XC6200 fastmap processor interface," Proc. 5th International Workshop on Field Programmable Logic and Applications, 1995.
6. Kwok: "An investigation of virtual hardware using FPGA technology," Honours year dissertation, Department of Computer Science, University of Edinburgh, 1996.
7. Lysaght, Stockwood, Law and Girma: "Artificial neural network implementation on a fine-grained FPGA," Proc. 4th International Workshop on Field Programmable Logic and Applications, 1994.
8. Wirthlin and Hutchings: "Implementation approaches for run-time reconfiguration," Proc. 5th International Workshop on Field Programmable Logic and Applications, 1995.