The Swappable Logic Unit: a Paradigm for Virtual Hardware

Gordon Brebner
Department of Computer Science
University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ
United Kingdom

Abstract

Swappable Logic Units (SLUs) were introduced in [2] to play a role in virtual hardware subsystems that is analogous to the role of pages or segments in virtual memory subsystems. The intention is that a conventional operating system can be extended to manage SLU circuitry implemented using FPGA real estate. In order to minimise operating system overheads, two particular SLU-based virtual hardware models were deemed practical in [2]: a 'sea of accelerators' model and a 'parallel harness' model. This paper looks in some detail at how SLUs will fit within the overall environment of a fairly conventional hardware/software system. First, there is a discussion of the FPGA-based hardware environment for SLUs, followed by a discussion of the software environment from which SLUs might be used. After this, there is a description of the operational properties that SLUs can have, and how these fit in with the two virtual hardware models. Finally, proposals for standard interfaces between SLUs and their environment are discussed. These interfaces can be regarded as constraints on the designers of SLU circuitry or, more positively, as suppliers of an enriched context within which such circuitry operates. The overall impact of the work presented in the paper is to show that it is feasible to incorporate configurable hardware within traditional computer systems that use high-level language programs and computer operating systems. That is, it should not always be necessary to devise special-purpose hardware/software systems to realise custom computing.

1 Introduction

In [2], the author described and demonstrated the components of a simple operating system able to support two virtual hardware models using physical FPGA resources, specifically the Xilinx XC6200 FPGA family [8,15]. Novel features of the XC6200, particularly partial reconfigurability and the ability to configure bus-mapped registers on the array, enable more sophisticated models to be implemented, compared with most other current FPGA technologies.

The central feature of the virtual hardware models was the **Swappable Logic Unit** (SLU). This is a logic circuit implemented using SRAM-based FPGA technology, and capable of performing a function in terms of specified input(s), and then supplying results as specified output(s). An SLU is the virtual hardware analogue of a page or segment in a virtual memory system. Two key features of an SLU, to facilitate its efficient handling by an operating system, 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. Note that, looking at the relationship between virtual memory and virtual hardware, a single memory location, or a block of memory, both fit within the definition of an SLU, so virtual hardware is a strictly more powerful concept.

In [2], various possible models for input and output arrangements for an SLU are described:

- signals on the perimeter of the SLU;
- bus-accessible registers within the SLU;
- 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 swappable 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.

This paper explores the nature of SLUs in more detail, focusing both on the implications for SLU circuit design and on the implications for interfacing SLUs to software executed by a conventional processor. The result is to indicate how an operating system can act as a familiar general-purpose intermediary in hardware/software co-designed configurable systems, as an alternative to the provision of special-purpose mechanisms for combining hardware and software components. However, the general approach has benefitted from the experience reported for rather more special-purpose systems (e.g., [5,10]), as well as for other types of general configurable systems (e.g., [4,6,14]). It is the intention that this paper supplies solid foundations for implementation work on SLU-based virtual hardware operating systems.

Before embarking on the study of SLUs, some assumptions about configurable systems, and the role of FPGAs in such systems, will be stated, to provide a context for this work:

1. configurable systems can provide performance benefits over purely software systems, for various applications;

2. it is desirable to harness the full capabilities of FPGA components used within configurable systems;

3. FPGA components have narrow interfaces to other hardware components;

4. it is desirable to interface FPGA components to software components using familiar computer system mechanisms;

5. any operating system used to manage this interface must impose only very small performance overheads.

The first point is particularly important, and must underpin any efforts to use FPGAs for building configurable systems, rather than just for building fixed logic circuits.

Experience has shown that careful selection of applications, and careful circuit design, is needed if significant performance gains are to be made from using configurable systems. The work of the PAM (Programmable Active Memories) group has provided excellent evidence of this [1]. Apart from the fact that normal processor chips are fine-tuned to provide better performance for operations on standard data types than FPGA circuitry can deliver, the third point above — the narrow (and relatively slow) interface to FPGA chips — is a bottleneck that can eliminate any performance improvement gained by transferring computation from a processor to an FPGA circuit. Competitive configurable systems must usually rely on the exploitation of pipelining and/or parallelism in the FPGA circuitry, and also the use of cache memory directly linked to the FPGA hardware. These facts must be borne in mind when thinking about SLUs and their management.

The fourth point above is important to this work. An aim is to allow software components of configurable systems to be produced using completely standard programming techniques, including the use of standard compilers. In principle, the programmer can be unaware that his or her software will be interacting with (virtual) FPGA circuitry. The assumption here is that hardware components (i.e., SLUs) will be designed separately, perhaps by the same person who is programming the software, but more likely not. Modest constraints must be placed on the design of such circuitry, to ensure that it forms a valid SLU — these are of a type that can be checked and/or satisfied automatically by design tools. A practical tool for this purpose is described in [3]. In summary, the essence of the approach here is that SLUs correspond to functions that are callable by computer programs. This approach is a less ambitious, but more practical, aim than allowing an entire hardware/software system to be described in a single high-level notation, with the automatic generation of hardware and software implementations as appropriate. The latter is, of course, a very interesting and active research area. However, the approach here is more likely to deliver better performance gains, in the medium term at least.

It is useful to standardise terminology in connection with the fifth point above, by distinguishing between operating systems and run time systems, and also highlighting the structure of operating systems. Here, an operating system is taken to be software that manages the physical resources of a computing system, while a run time system is taken to be software that manages the logical resources of a program being executed. In most current FPGA-related environments, the operating system just supplies a device driver to allow access to the FPGA hardware, and a run time system is able to do everything else, since the FPGA hardware is dedicated to a single program. Here, the operating system has a rather larger part to play. Its kernel manages the FPGA resource, just as it manages the physical memory resource. In particular, several executing programs might be allowed to share use of the FPGA hardware. Note that operating system components outside the kernel might make use of virtual hardware too; such use is just a special case of use by run time systems. This paper is not concerned with how operating systems might be enhanced by FPGA use to best effect, although this is also a very interesting research question in itself.

The organisation of the paper is as follows. First, Section 2 discusses the hardware environment in which SLU circuitry must operate, and Section 3 discusses the software environment from which SLU circuitry must be harnessed. Then, Section 4 considers the basic operational properties that SLUs can have, and how these properties can be exploited in the two SLU-based virtual hardware models. Following this, uniform input/output interfaces for SLUs are discussed in Section 5. Some final conclusions are drawn in Section 6, together with an outline of experimental work that is planned in order to build on the foundational ideas contained in this paper, to demonstrate practical feasibility.

2 Hardware environment of an SLU

An SLU must operate within the hardware environment of an FPGA. Thus, the circuitry of SLUs is described by configuration data for the particular FPGA technology being used. However, the design of SLU circuitry does not need to be technology-specific, unless it is intended to harness specific technological features. The essential characteristics of an SLU — for operating system purposes — are its area and its input/output interface, and this information could be provided either by the technology-dependent phase of a design tool, or by automatic analysis of configuration data produced by such a tool. The latter approach is illustrated in [3], as a way of carrying out SLU implementation using existing hardware design tools. This is harmonious with the policy of also using existing software design tools without modification.

For some FPGA technologies, for example, the Xilinx XC6200 family, extra information about the structure of SLU circuitry can be invaluable for enhancing the performance of a virtual hardware operating system. In [2], two particular types of structural information were identified: replicated patterns and parameterised areas. The first is concerned with exploiting repeated patterns of cells within an SLU, so that an SLU can be swapped in more quickly by parallel configuration of the repeated parts. The second is concerned with exploiting 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 explicitly designed, since these can be instantiated with particular cells in their parameterised areas. Both of these matters exploit features of the XC6200 FPGA: parallel configuration and partial reconfiguration respectively. Although the two concepts have more generality, their specific implementation relies intimately on the actual FPGA implementation of SLUs, and illustrates a matter in which an operating system must be tailored to the available hardware in order to improve its operational efficiency. As with the basic SLU information, it is feasible for such structural information to be derived from high-level, technology-independent, circuit descriptions. The use of a hardware description language, such as Ruby, Verilog or VHDL, is more appropriate for this purpose, rather than schematic capture.

In the simplest situation, an SLU can be accommodated within a single FPGA chip, and this chip interfaced to a conventional processor-memory subsystem as an extra memory device. (Necessarily, the FPGA technologies used in this work are assumed to be SRAM-based.) In this case, configuration and use of the SLU are constrained by the pins available on the chip, and the way these are interfaced to the main hardware subsystem.

More generally, the SLU may be contained in more sophisticated hardware environments:

- arrays of multiple FPGA chips; or
- arrays of FPGA chips directly interfaced to local memory.

The first class of environment deals with the problem of limited gate counts on single chips, and the second class with problems of bandwidth between FPGAs and system memory. A further elaboration would be the addition of a slave processor to manage the FPGA subsystem, although normally only an FPGA is provided as a controller.

Multi-chip arrays introduce problems, since pin limitations are unlikely to allow the inter-chip gate connectivity to be as rich as the intra-chip gate connectivity. Further, the use of off-chip signals will reduce performance. The first point is a factor that may reduce position independence of an SLU. An SLU may only be placed in an arbitrary position on a multi-chip area if its circuitry is constrained to respect limitations imposed at chip boundaries. The second point means that, if position independence is achieved, performance when using the SLU may be adversely affected, depending on its placement. Of course, these difficulties can be avoided when SLUs are small enough to fit on a single FPGA chip (possibly compromising the operating system's scope to use the total space of a multi-chip array efficiently). This matter is related to considerations of granuality and optimal SLU size.

The addition of local memory to an FPGA subsystem need not directly impose extra problems for SLU circuitry. Ultimately, this matter is covered within the relationship between an SLU's input/output interface and the input/output pin interface of the SLU's host FPGA chip (or the perimeter input/output interface of its host multi-chip array). However, it seems wise to make provision for local memory interaction when defining a standard SLU interface, since cache memory is frequently needed to make configurable hardware efficient. Thus, this factor is taken into account in some of the interfaces proposed in Section 5.

3 Software environment that can use an SLU

The operation carried out by an SLU is intended to be analogous to that carried out by a function in software. That is, inputs to an SLU correspond to function parameters and outputs from an SLU correspond to function results. The exact correspondence depends on the programming paradigm being used. Examples of different paradigms include:

- imperative programming (e.g., C, Fortran);
- functional programming (e.g., LISP, ML);
- logic programming (e.g., Prolog);
- object-oriented programming (e.g., C++, Java);
- event-driven programming; (e.g., using C with the Windows API);
- parallel programming (e.g., Occam, Parallel Fortran); and
- machine code programming (in assembly languages).

The idea of functions, and calls on functions, is universal to these paradigms, although the overall approach to programming differs. For the first three paradigms, SLUs can be regarded as direct substitutes for library functions supplied to the programmer. In the case of logic programming, there is extra scope for configurable hardware involvement in implementing the overall inference engine, but this matter is not considered here as a separate issue. For object-oriented programming, a related set of SLUs can correspond to the set of methods (operations) provided by an object class.

For event-driven programming, functions with the normal call/return behaviour exist to control and handle events. However, the effect of using the event-driven paradigm is to allow 'functions' with separated calling and returning. That is, calling involves starting one or more activities that will lead to future events, and returning involves later handling of one or more events that result from the activities. In SLU terms, this corresponds to supplying inputs, and then collecting outputs at some later time, without busy waiting in between for the circuitry to compute.

For parallel programming, there are constructs that allow parallel execution flows to be specified, either in a SIMD or MIMD style. The SIMD model essentially allows one function to be executed multiply in parallel, and the MIMD model essentially allows a collection of functions to be executed in parallel. Provision also has to be made for communication between parallel execution flows. In SLU terms, this communication means that some operating system support must be given for inter-SLU connectivity.

For programming in machine code, a subroutine mechanism is available, and this can be used in exactly the same way as functions in higher-level programming paradigms. More interestingly, machine code instructions can be viewed as 'functions' here, since they have a similar input/output behaviour, albeit with a far finer grain of operation. The idea of using FPGAs to support virtual instruction sets for processors is already established and demonstrated, for example, by the DISC project [13]. Note that the activity of machine code programming also includes the work of compiler writers, who must determine the machine code that is generated from higher-level language programs.

In the case of the high-level paradigms (i.e., all except machine code programming), there are two ways in which the interface to an SLU can be presented to the programmer. The first way is to make the operation of the SLU available as a single library function with appropriate parameter and result types. This style is very familiar to programmers, since library functions are a universal feature in computer programs. The interface can be supported using standard run time system and operating system mechanisms.

The second way is to make a compiler responsible for introducing SLU operations to support built-in programming language features. This requires extra work to be done on tailoring compilers, in addition to the operating system work that must be done to support virtual hardware.

In the first instance, it seems reasonable to make SLUs available using just standard library function interfaces. Such an approach is also used in the parallel computer community where, for example, the MPI (Message Passing Interface) package [11] for system-independent inter-processor communication has become widely used as a library of functions called from normal C (or C++) programs. In the overall context of this work, tailored compilers and instruction-level SLUs are a worthy future topic of investigation after the basic SLU mechanisms have been well tested.

4 SLU operation, and virtual hardware models

SLU operation

In [2], a simple type of SLU was used for experimental purposes; this type can be described as the *stateless* and *passive* SLU. In other words, the SLU is a simple piece of combinatorial logic that transforms circuit inputs into circuit outputs. In programming terms, this corresponds to a function that does not affect global variables (and has no local variables that retain their values over successive function calls). Two enhancements to the simple SLU are possible, and this affects the interface between an SLU and its environment, which is the topic of Section 5.

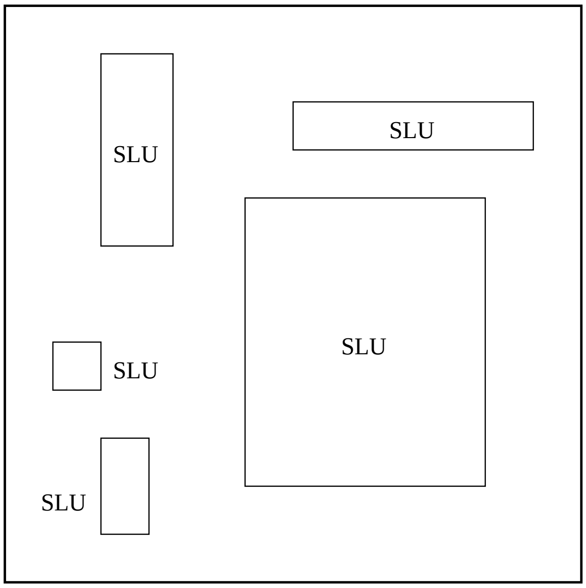
A first enhancement is to make SLUs *stateful*, that is, to introduce sequential logic including latches or flip-flops. In programming terms, this corresponds to a function that can retain variables across successive function calls. Such functions are common, and so the stateful SLU may be meeting software requirements. However, the introduction of state may be for other reasons, notably where pipelining is used within the SLU in order to improve performance. In this case, a simple call-return model for functions is no longer adequate. One possibility is to supply a vector of parameters to a function call, and to collect a vector of results, one result per parameter. The SLU then operates on the parameters as successive inputs, with the results emerging as successive outputs. This means that a more complex interface to SLUs is needed, to cope with vectors and with the associated timing matters. An alternative is to adopt an event-driven programming style, where the provision of parameters and the collection of results is separated in time. This could still use the simple stateless SLU interface, which makes life easier.

A further issue introduced by stateful SLUs is that the virtual hardware management may need to include swapping out as well as swapping in. It is not necessary to swap out the entire SLU circuitry, just the parts that contain state information. The basic apparatus for identifying these parts already exists: SLUs can have parameterised areas, designed to support partial SLU reconfiguration. Here, there is a similar idea, the difference being that areas are selected for swapping out rather than for swapping in. In essence the stateful circuitry is a very dynamic form of parameterised circuitry.

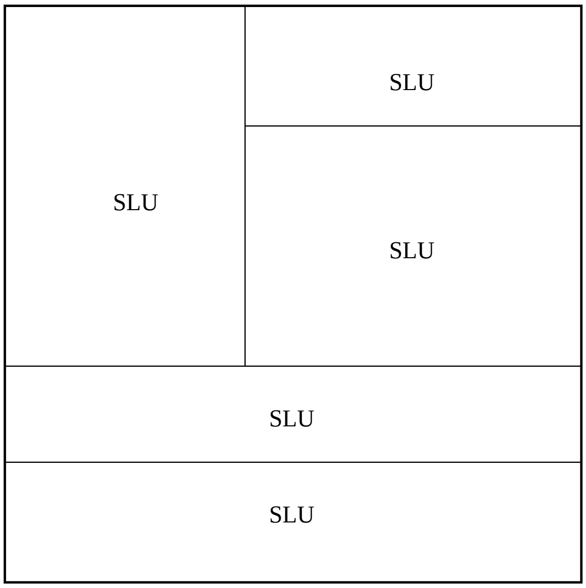
A second enhancement is to make SLUs *active*, that is, to allow the generation of outputs that are not directly stimulated by new inputs. In programming terms, this corresponds to background activities that run in parallel with the main flow of program execution. These are natural consequences of an event-driven programming style. At a lower level, interrupt-driven peripheral devices, possibly involving DMA memory access, are a familiar example. Active SLUs require a more complex control interface with the rest of the overall system, to allow more than just taking inputs from a processor and making outputs available to a processor.

SLU-based virtual hardware models

Two SLU-based virtual hardware models were suggested in [2], both intended to allow very efficient support from operating systems, by keeping dynamic placement and routing extremely simple. One is the *sea of accelerators* model, consisting of a collection of independent SLUs resident on a configurable array. Two examples are shown in Figure 1. In the second example, although the SLUs are adjacent to one another (and to the perimeter of the sea), note that there is no communication across SLU boundaries. The other model is the *parallel harness* model, consisting of a collection of cooperating SLUs embedded within regular inter-connection wiring on a configurable array. Two examples are shown in Figure 2. In the second example, connections between SLUs are achieved just by juxtaposition, and no extra wiring is needed. Note that, for parallel systems with irregular interconnection patterns and relatively infrequent communication, then the first model, with a processor

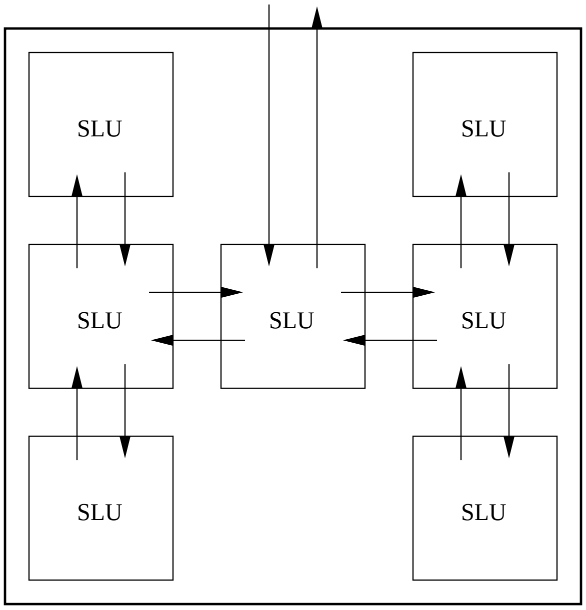


(a) Sea with five accelerators present, placed fairly randomly

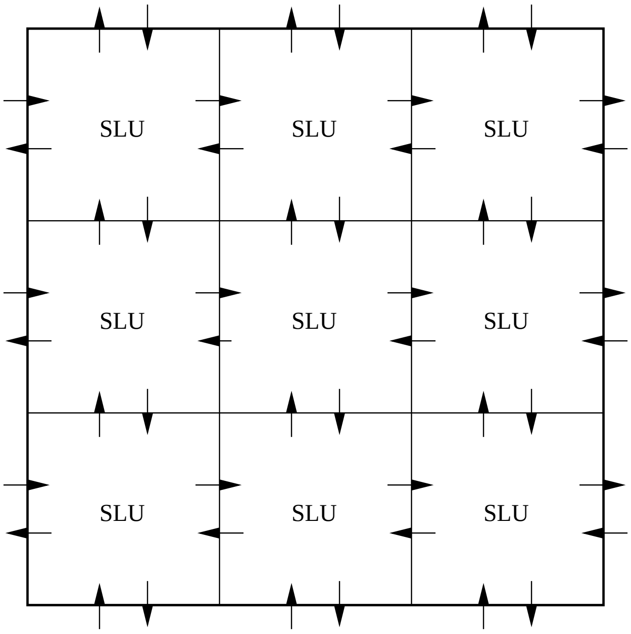


(b) Sea with five accelerators present, densely packed

Figure 1: Two examples of configured seas of accelerators



(a) Parallel harness with extra H-tree wiring supplied



(b) Parallel harness with no extra wiring needed

Figure 2: Two examples of configured parallel harnesses

acting as a sequential communication device between SLUs, is a better solution than the second.

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 efficiently managing a large collection of small circuits and as a way of managing a circuit composed of a collection of regularly-connected small circuits.

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 difference is that the parallel harness model is an abstraction of parallelism at execution time, whereas SLU pattern identification is to aid parallelism at configuration time.

In general technological terms, the sea of accelerators model is intended for use with SLUs that have bus-addressible register interfaces as part of their circuitry, rather than signal interfaces on their perimeter. This allows an operating system to take responsibility only for placement, with routing not being required. This is indicated in Figure 1 by the fact that no explicit connections to the outside world are shown. The parallel harness model is intended for use with SLUs that have signal interfaces on their perimeter, designed for interaction with neighbouring SLUs. The operating system supplies fixed, regular, routing between SLUs, for example a rectangular grid or a binary tree. It also supplies routing from FPGA input/output pins to the perimeter of the parallel harness. Of course, one special case is where there is a single SLU: this covers the traditional practice of having a single resident virtual circuit with perimeter interaction.

The sea of accelerators model is the closer to conventional virtual memory, the essential difference being that it involves two-dimensional, rather than one-dimensional, placement. It can be seen as the software-driven model. The parallel harness model is rooted in various FPGA-related approaches to virtual hardware. It can be seen as the hardware-driven model. There is a place for both models when planning configurable systems. Currently, the first model is only well-supported by one technology known to the author (the Xilinx XC6200), but this will change as FPGA technology evolves to reflect the needs of configurable systems. The latter model is supported by all SRAM-based technologies to some extent; partial reconfigurability is desirable, to allow modification of specific SLUs with the parallel harness, and this is supported by some FPGAs, for example, the Atmel AT6000 as well as the Xilinx XC6200.

Regardless of which model is chosen, the interface to constituent SLUs must ultimately be implemented using perimeter pins of the FPGA(s) used. In the case of the sea of accelerators model, the relevant pins are those for the SRAM interface. Assuming that this is a single-ported interface, then it acts as a bottleneck between the SLUs and their environment; in particular, only one SLU can be actively interacting at a time, and only for input or output, not both simultaneously. In the case of the parallel harness model, there is more flexibility in the input/output arrangements. However, this does not extend to routing input/output signals to or from arbitrary SLUs. Only perimeter SLUs have external access, and so the model is best suited to parallel architectures like systolic arrays.

A further physical influence on constituent SLUs is the existence of global signals for an FPGA, notably clear and clock signals. A global clear signal can be used selectively by an operating system for initialisation and re-initialisation of the entire virtual hardware environment. A global clock signal exerts an influence on any SLUs containing sequential logic, and so may introduce dependence between SLUs that are notionally independent, and also influence the behaviour of SLU interfaces. Given that all presently-available FPGA technologies are synchronous, clocks and timing are an important issue that cannot be ignored in virtual hardware models. This is particularly the case when an invisible interface to software is required. The SLU interface described in the next section takes account of this concern. For the future, asynchronous FPGA technologies, which are currently the subject of research (e.g., [7,12]), offer much promise for virtual hardware including, here, enabling true independence of SLUs as swappable entities.

5 Uniform interfaces for SLUs

From the point of view of a hardware designer, the ideal is to be able to construct SLU circuits, exploiting the full range of capabilities of the design environment used, and the full range of capabilities of the FPGA technology being used for implementation. Usually, some compromises have to made between these aims in any circuit design. However, for SLUs, the need for compatibility with a virtual hardware environment imposes additional constraints. Preceding sections have alluded to some of these constraints. This section takes them into account in suggesting appropriate standard interfaces for SLUs. Turning the argument round, these interfaces can be seen as providing supportive environments for SLUs, rather than compromising SLU designers.

Given the virtual hardware models proposed, there are three different classes of SLU that are of interest:

- accelerator SLUs (used within the sea of accelerators model);
- PE (Processing Element) SLUs (used within the parallel harness model); and
- system SLUs (used in support of a model, or as 'raw' circuitry).

The relationship between an SLU and its environment can be viewed as having three levels, from bottom to top:

- the physical interface between the SLU and its environment;
- the bit-level interaction between the SLU and its environment; and
- the functional interaction between the SLU and its environment.

The exact nature of these levels varies according to SLU class.

Physical interface

In terms of the physical interface, accelerator SLUs have a register-style interface, PE SLUs have a perimeter port interface and system SLUs have an interface that uses any available mechanisms on the host FPGA. As mentioned earlier, the operating system needs to know the locations of these interfaces on each SLU used. For any SLU type, this can be specified as a list of input positions and a list of output positions. Each position is a list of bit coordinates, either register bit coordinates or perimeter bit coordinates. These coordinates are relative to the SLU for accelerator or system SLUs (i.e., the bottom left of the SLU is at position $(0, 0)$), and are absolute for system SLUs (i.e., relative to the host FPGA(s)).

In addition to these 'local' interfaces, global signals may be available to an SLU, as well as to all other SLUs contained in the virtual hardware. In the case of accelerator SLUs, the only type of global signal that fits with the spirit of the sea of accelerators model is a clock signal, certainly to clock the register interface, and possibly for use in clocking other sequential circuit components or as a general time source for the circuit. The effect is that all accelerator SLUs present in the virtual hardware would be synchronised to a system clock. In the case of PE SLUs, a more extended role is possible for global signals, in providing broadcast information to all PEs in a parallel harness. Such signals can then be specified as part of the overall harness provision. In the case of system SLUs, arbitrary use of available global signals is legitimate.

Bit-level interaction

For an accelerator SLU, the lowest level interaction forces the SLU to be passive, since its interface registers are essentially just memory devices to a surrounding hardware environment. Elaborations, such as treating the registers as being more like control and status registers of an input/output device, are a matter for higher level interactions, considered in the next subsection.

For a PE SLU, the lowest level interaction is with the parallel harness, and this is merely a routing device to reach neighbouring SLUs. These may be other PE SLUs in the harness or system SLUs at the perimeter of the harness. In either case, the lowest level interaction is just the input and output of bit signals. Elaborations, such as the encoding of higher-level information in terms of bit signals, are a 'user' matter for the overall parallel SLU system designer.

For a system SLU, the lowest level interaction can be richer, since the SLU is able to interact directly with the surrounding environment of the host FPGA. As well as external devices connected to the FPGA, direct access to (sub)system buses is possible. Through bus interfaces, a system SLU could be made capable of direct memory access or raising processor interrupts, for example. Depending on the overall hardware architecture, this will affect local subsystem resources if they exist, overall system resources otherwise.

Functional interaction

One aspect of this level of interaction is concerned with allowing accelerator SLUs to behave in a similar style to system SLUs, but in a protected manner. In general, the natural way to define an appropriate high-level interface is to model chip input and output pins using registers. For example, one clock bit input pin can be modelled by a one-bit register read by the SLU; 16 address bit output pins can be modelled by a 16-bit register written by the SLU. This approach relies on a processor reading and writing such registers, and interpreting their contents appropriately. The software for this would be supplied by the operating system, to support the defined high-level SLU interface. In turn, the operating system would liaise with the software that is using the SLU, via the software's run time system.

Another aspect of this level of interaction concerns a feature of use to all types of SLU. This is the reconfiguration of parameterised SLU areas. It can be done under software control without the need for changes to the SLU interface. However, an alternative (which has no analogue in traditional virtual memory) is for SLUs to initiate their own reconfiguration. This could be implemented from first principles using the mechanisms available to accelerator and system SLUs (and globally for PE SLUs, via surrounding system SLUs). A higher-level scheme seems desirable, but no detailed proposals will be presented here, pending experimental investigation. The business of enabling an SLU to indicate which parameterised areas should be updated, and with what, seems to be complex potentially, if it is to be provided as a general-purpose facility.

Finally, an aspect relevant to both accelerator and PE SLUs is the provision of a mechanism for supplying sequences of inputs and collecting sequences of outputs. This is of use, for example, in pipelined accelerator circuitry and in systolic array circuitry. Such a mechanism need not act as a further constraint on SLU design; rather, it gives operating system support for particular design styles. A simple implementation would just involve the operating system (running on a processor) doing the necessary sequenced work, ensuring that it respects the SLU input/output clock rates. An alternative is to add support for managing an interface between local subsystem memory and SLU circuitry.

6 Conclusions and future work

The approach proposed in Section 3 is to make the capabilities of SLUs available as library functions that can be called from software programs. This raises two fundamental questions that lie at the heart of practical hardware/software co-design:

- how complex should individual functions (and hence SLUs) be?
- when should a function be computed in hardware using an SLU?

This paper does not seek to provide answers to these questions, rather to give a general framework within which the issues raised by the questions can be investigated. The SLU concept is very scalable, from small SLUs at the processor instruction level, to large SLUs for complex functions. The library function software interface to SLUs gives scope for run time systems to make dynamic choices between hardware and software implementations of the same function, for example, based on program profiling.

A further question concerns the practicality of (possibly frequent) swapping of SLUs in a virtual hardware system. This has been addressed by the choice of virtual hardware models introduced in [2], and then placed in an overall hardware/software context by this paper. A major underlying force is to keep matters simple, and so to minimise overheads. However, this has not compromised the overall goal: the details given here of SLU behaviour and interfacing address the effective harnessing of FPGA technology: a unique blend of processor, memory and input/output devices.

Further work in the immediate future will involve experimentation. In [2], basic experiments with primitive SLUs and primitive environments helped to establish the general concepts, and prove that it was practicable for operating systems to manage collections of SLUs. This paper has refined the ideas to a point where further implementation and experimentation are necessary to demonstrate that practical virtual hardware systems can be built, to bridge the gap between high-level software programs and FPGA circuitry in a seamless manner. The work will be carried out using Xilinx XC6200 chips contained within a subsystem containing local memory, and interfaced to a system bus. One such subsystem is described in [9]; other, commercial, subsystems are also available from Annapolis Micro Systems and Virtual Computer Corporation.

From the software point of view, the first step will be to use existing device drivers available for the Linux and Windows operating systems, and place all other functionality in the safe context of a run time system. Once this is working effectively, the management of the FPGA resource will be moved down into the operating system. For this work, Linux will be used, due to the easy availability of source code. It is also hoped that these issues can be investigated within the framework of the Oberon system developed at ETH, Zurich. No major problems are expected in producing run time system and operating system components that can carry out the support required by SLUs and the virtual hardware models. The hard challenges lie beyond the system software development: ensuring that genuine performance gains are achieved in working hardware/software systems.

7 References

[1] Bertin, Roncin and Vuillemin, "Programmable active memories: a performance assessment", Proc. 1993 Symposium on Integrated Systems, MIT Press 1993, pp.88–102.

[2] Brebner, "A virtual hardware operating system for the Xilinx XC6200", Proc. 6th International Workshop on Field Programmable Logic and Applications, Springer 1996, pp.327–336.

[3] Brebner, "Automatic identification of Swappable Logic Units in XC6200 circuitry", submitted to 7th International Workshop on Field Programmable Logic and Applications, London, September 1997.

[4] Casselman, Thornburg and Schewel, "Hardware Object programming on the EVC1 - a reconfigurable computer", Proc. FPGAs for Rapid Board Development and Reconfigurable Computing, Photonics East 95.

[5] Eldredge and Hutchings, "Density enhancement of a neural network using FPGAs and run-time reconfiguration", Proc. 2nd IEEE Workshop on FPGAs for Custom Computing Machines, IEEE 1994, pp.180-188.

[6] Hadley and Hutchings, "Design methodologies for partially reconfigured systems", Proc. 3rd IEEE Workshop on FPGAs for Custom Computing Machines, IEEE 1995.

[7] Hauck, Borriello, Burns and Ebeling, "MONTAGE: an FPGA for synchronous and asynchronous circuits", Proc. 4th International Workshop on Field Programmable Logic and Applications, Springer 1994.

[8] Kean, Churcher and Wilkie, "XC6200 fastmap processor interface", Proc. 5th International Workshop on Field Programmable Logic and Applications, Springer 1995, pp.36–43.

[9] Ludwig, "The design of a co-processor board using Xilinx's XC6200 FPGA — an experience report", Proc. 6th International Workshop on Field Programmable Logic and Applications, Springer 1996, pp.77–86.

[10] 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.

[11] MPI Forum, *MPI 1.1: A message-passing interface standard,* MPI Forum 1995.

[12] Payne, "Self-timed FPGA systems", Proc. 5th International Workshop on Field Programmable Logic and Applications, Springer 1995, pp.21–35.

[13] Wirthlin and Hutchings, "A dynamic instruction set computer", Proc. IEEE Workshop on FPGAs for Custom Computing Machines, IEEE 1995.

[14] Wirthlin and Hutchings, "Implementation approaches for run-time reconfiguration", Proc. 5th International Workshop on Field Programmable Logic and Applications, Springer 1995, pp.419–428.

[15] Xilinx Inc., XC6200 advance product specification, Xilinx 1997.