# Circlets: Circuits as Applets

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

## Abstract

*Custom computing is concerned with deriving benefits from importing the flexibility of (software) programs into (conventionally, hardware) circuitry. A main aim is to gain speed-ups by programming at a level closer to the physical hardware, and in a medium that allows explicit parallelism. Currently, this goal is hampered by the simplistic ways in which configurable circuitry is interfaced to other computer system components, but this will improve with the evolution of system-on-chip technologies. The new flexibility offers other benefits too, one example being adding fault tolerance to systems. This paper is concerned with making the first steps towards another benefit: portability of circuitry in a network computing environment. That is, expressing applets in circuitry terms, rather than program terms. The work draws on earlier research of the author into low-level circuit specification and into virtual circuitry. An implemented client-server system is described, and then this is followed by discussion of how* circlets *— a term introduced in this paper to denote applets expressed as circuits — might best be represented in a portable form, and also how circlets may be supported by appropriate execution environments. At this point in history, when designers of FPGA technologies are still exploring the space of technical possibilities, it is not possible or desirable to attempt a definitive specification of circlets and their management. However, it is apt to study the problems posed by circlets, since this should influence future directions for FPGA technological development.*

## 1 Introduction

Configurable logic circuitry offers an exciting opportunity to probe trade-offs between speed and generality, by blurring the line between fast but fixed hardware and flagging but flexible software. The blurring has been sufficient for this author, at least, to become very careful about the use of the words 'hardware' and 'software' — specifically, to restrict the former to fixed physical components, and to use the latter for all fluid logical components[1] (this is consistent with a distinction made between atoms and bits, as popularised by Nicholas Negroponte [11]). In particular, configurable logic circuitry is a form of software, the key difference from the norm being that it is expressed in circuit form, rather than in program form. This is what raises the interesting questions about its use. Comparing circuits with programs, there are two likely benefits to be gained. The first is that circuits are likely to involve relatively simple components and also to be relatively trivial to implement using real hardware. The second is that most circuits include parallelism in the execution of functions, whereas most programs do not. In summary, these mean higher execution speeds, and a reduced system support requirement.

At the current stage of development, speed benefits from the use of configurable logic circuits have often been elusive. Even where there are apparent gains, it may be the case that devoting as much effort to program development as is usually necessary for circuit development can significantly diminish any gains (e.g., [13]). A major reason for lack of progress is that the FPGA technologies used to implement configurable logic circuits are packaged as separate chips, and often isolated on separate boards, with the result that narrow and slow interfaces destroy speed gains from the circuitry. The author is confident that this situation will improve markedly with the rise of System-Level Integration, when configurable logic will be integrated on chip with things like processor cores and memories, and indeed will be inserted into processor cores

[1] One consequence is that the author now prefers the term 'virtual circuitry' to 'virtual hardware' — there is a possible confusion with the term 'virtual circuit', which is widely used in computer networking, but it seems worth the risk.

and memories. Pending these enabling technological advances, this paper is concerned with preparing the ground, by considering circuits as a possible alternative to programs when used as portable components of a distributed computing system.

The client-server model has been pervasive in distributed computing for two decades. It represents the rise of computer networking and the demise of the single all-functions computer. The basic idea is that one computer — the client — can make use of functions made available by another computer — the server. In its initial incarnation, the model generally carried a presumption that a client computer was a cheaper user workstation, whereas the server computer was a more expensive, shared and specialised system component. Examples include file servers, printing servers and authentication servers — computers equipped with specialist hardware or software. In more recent years, this presumption has been shaken, notably by the emergence of WIMP servers, where the human-interaction facilities of a workstation provide a service to client computers elsewhere. This trend has now widened with the emergence of network computers, where execution and storage facilities of a workstation provide a service, exercised using applets. Note, however, that the workstation may still be viewed as being a client, with the supplier of applets being the server — there are no definitive clear-cut roles.

Configurable logic circuitry is a natural candidate as a server component. It offers scope for accelerating computations, while having flexibility derived from programmability. At the present time, such reconfigurable processing units are mostly in the style of the traditional shared and specialised remote server computer. In the future, these units will become servers in the local workstation, as technologies and concepts mature. The latter can learn many lessons from the former, both in terms of how logic circuitry on one computer can be effectively harnessed from another computer, and also in terms of how configurable logic circuitry can be effectively integrated with other computer system components.

This theme is developed in the next three sections of the paper. In Section 2, a novel implemented client-server system for configurable logic, based on the shared use of a remote server, is described. The description includes comment on the general lessons to be learnt from this particular system. Section 3 introduces the idea of a *circlet* — a configurable circuit form of an applet — and considers how it may be represented in a portable fashion. After this, Section 4 examines the execution environment that a computer system might provide for an circlet, drawing upon the author's previous work on virtual circuitry. Finally, Section 5 contains some conclusions, and points to future applications of circlets.

## 2 SLURP: the Swappable Logic Unit Remote Procedure server

The notion of a shared remote configurable logic server is not new. However, existing examples are pitched at a low level of abstraction. That is, a server contains one or more boards carrying FPGA chips, and then the network interface essentially allows a client to emulate the system bus interface to the board. This makes the client-server system both FPGA-chip and board specific, which runs contrary to the normal machine-independence associated with client-server systems. As an example, the Webscope system developed by Xilinx Development Corporation allows direct read and write access to the SRAM of XC6200 FPGAs, as well as to signals for controlling such things as clock and reset lines. Not only is this device-specific, but it also gives control of a whole board to the client user.

The SLURP (Swappable Logic Unit Remote Procedure) server raises the level of abstraction somewhat, by focusing on the Swappable Logic Unit (SLU) concept introduced in earlier work of the author on virtual circuitry. There, an SLU was a circuitry analogue of a page or segment in a virtual memory system, and roughly corresponded to a function callable from within a program. Here, it is used as a circuitry analogue of an applet in a network computing system and, once sent to the server, roughly corresponds to a remote procedure callable from the client. Expressed in this way, it is fairly clear that many of the management mechanisms developed for virtual circuitry are immediately applicable to the new problem; the major new feature is that the server and client are distinct.

The SLURP server runs under Linux, on a PC equipped with a Virtual Computer Corporation PCI-XC6216DS board. The board, which carries one Xilinx XC6216 chip [14] and is interfaced via the PCI bus, is accessed using a Linux driver developed at Brigham Young University. It is managed using the author's prototype SLU-based operating system that supports a 'sea of accelerators' model, by allowing the dynamic placement of independent SLUs which have mapped-register input/output interfaces. This system was described in [1], when it was based upon an XC6200 simulator; the Linux version used for this work is the first one to use the real hardware. SLURP clients have been developed both for Linux PCs and for Sun workstations, ensuring portability across different byte-sexing arrangements. There would be no major technical problems involved in making both server and client software available on non-Unix platforms, save personal disinclination on the part of the author.

In essence, SLURP makes the low-level operating system functions available remotely. Thus, there are three core operations available to the client:

- add a new SLU to the collection known at the server (and remove a known SLU if no longer required);
- instantiate a known SLU on the logic array at the server (and wipe out an instantiated SLU if no longer required); and
- execute the SLU function with supplied inputs, and return the outputs.

There are also housekeeping operations for opening, closing and aborting connections between the client and server. The overall effect is that a program on the client computer can make use of the same SLU operating system functions as are available on a computer equipped with FPGA resource. The server allows more than one client to be connected simultaneously.

Communication between the client and server takes place using a specially-designed SLURP protocol. This runs on top of a standard TCP/IP connection and thus, in principle, means that a server can be accessed from anywhere on the Internet. The SLURP protocol is a request-response protocol, the client making requests and the server returning responses. There is a single SLURP protocol message type; however, it has a variable-format component that depends on the particular type of request or response. The only feature of the protocol that is not request-response is that either the client or server is able to send an abort message at any time, in order to unilaterally end the SLURP connection. Rudimentary security is included by the client sending a username and password when beginning a SLURP connection, and the server returning an integer key to be used in all future SLURP protocol messages. In a more hostile environment than that used for the SLURP prototype, rather stronger security features would be needed.

Swappable Logic Units are characterised by the following information, which represents a pragmatic compromise between XC6200-specificity and generality:

- two-dimensional bounding box for SLU;
- two-dimensional placement constraints for SLU;
- configuration data for SLU; and
- positions of input and output registers within SLU.

As reported in [4], it is possible to derive most of this information semi-automatically from raw configuration data for the SLU circuitry. It is envisaged that future SLU-oriented design tools will generate this information automatically.

The bounding box assumes that rectangular spaces are allocated for SLU placement, in order to increase efficiency. This is a technology-independent feature, although care has to be taken when determining exactly where the boundaries of a piece of circuitry are [4]. In contrast, the placement constraints only have a technology-specific motivation. They allow the SLU owner to specify fixed column or row address bits to be used when the SLU is placed on the logic array. For example, in the case of the XC6200, reasons include the need for placement on a four-cell or 16-cell boundary, or for placement to allow use of the wildcard registers. This reflects the fact that the description of the SLU circuitry is expressed as XC6200 configuration data, which is very efficient — but only when XC6200 technology is available to run the circuitry. The data are in the form of address/data pairs for the XC6200 logic array; the addresses are relativised by the operating system to take account of its placement decisions.

The input and output registers of the SLU are assumed to be oriented within columns — reflecting the XC6200 technology, which is asymmetric in this respect. Also reflecting the technology, the cells forming a register do not have to be in adjacent rows. Thus, the register description specifies a base row for the register within the SLU, and also a bit mask specifying the register cells relative to this base. The information is enough to allow the operating system to supply inputs and collect outputs using the bus-mapped register interface of the XC6200 [10].

A client asks the SLURP server to add an SLU to its collection by sending a descriptor containing the SLU information. The server checks the description thoroughly, to ensure that it makes sense and does nothing illegal. For example, the configuration data and register specifications may only refer to the area within the bounding box, and the placement constraints must be achievable. If the SLU is acceptable, the server stores it, and returns a unique SLU identifier to the client, for use in future operations. One of these operations is that to request the removal of the SLU from the server's collection — this is used automatically when a client-server connection is ended.

In order to make an SLU executable, a client asks the SLURP server to instantiate it on the logic array. Any SLU may be multiply instantiated, space permitting. The management of the collection of instantiated SLUs is a key operating system function, as discussed in [1], and is essentially a two-dimensional version of traditional one-dimensional page and segment management in virtual memory systems. If it is possible for a copy of the SLU circuitry to be placed on the array, the server returns a unique active SLU identifier to the client for use in future operations. One of these operations is that to request the wiping out of the SLU from the logic array — this is used automatically when an SLU is removed from the server's collection of known SLUs. An object-oriented analogy to the process of registering and activating SLUs at the server is that adding an SLU to the server's collection corresponds to declaring an object, and then activation corresponds to creating an object instance. Here, of course, the object is a rather simple, one-method, object where the method corresponds to the SLU's function.

Once an SLU has been instantiated, the client can make use of its execution capability. This involves a remote procedure call style of operation. The client sends an array of unsigned integers, one for each SLU input register, and receives back an array of unsigned integers, one from each SLU output register. Optionally, the client can also send a delay value, to specify a waiting period between supplying the inputs to the circuitry and collecting the outputs. This style of access to the SLU function allows integration with the run-time systems of most high-level programming languages, as discussed in [3]. Of course, this prototype mechanism is not claimed to offer dramatic function acceleration for clients. Given that the performance of the circuitry is already hampered at the server end by the PCI interface, the networked access only makes things worse. However, it illustrates that a client is able to use a specialised resource which it would not otherwise have access to. For example, the SLURP system has been used to allow users to develop XC6200 circuitry without any need to use a computer that contains any XC6200 hardware — a classic client-server application.

Although XC6200-specific to some extent (perhaps inevitably, given the pioneering nature of this technology with such things as bus-mapped registers), SLURP illustrates a higher level of operation from other remote-access systems. By focusing on SLUs, it removes the need for complete control of any board or chip; more than this, it makes no reference to the specific chip type or board type used at the server. Further, it denies access to such things as board control signals and chip input/output pins. All of this is consistent with usual practice for other hardware devices, where an operating system takes responsibility for their direct management. The main XC6200 specificity arises from the fact that SLU circuitry is described directly in terms of XC6200 configuration data; this is a matter that needs to be addressed if the SLURP server is to be generalised to other technologies, and is a central concern of the next section.

As a final point, note that the general SLURP mechanism can be used for the different client-server arrangement that would occur in a network computing environment. Here, it can be shrunk just to the SLURP capability for communicating SLU descriptions. A computer equipped with appropriate hardware would request an SLU from a remote computer, and it would respond by transmitting the SLU's description; in this case, the requesting computer is a client and the supplier is the server. After the SLU has been acquired, it can be managed by the local operating system, without any future network interaction. This explains why the next section focuses on the area of SLU description.

## 3 Representation of circlets

'Circlet' is a new term introduced in this paper, to mean a circuit that can be used as an applet. Just as a program-style applet can interact with its local computing environment, so a circlet is a circuit that can interact with its environment. However, the scope of this interaction will necessarily be somewhat simpler than that associated with programs. Any representation of a circlet must include a description of the circuitry and a description of relevant properties of the circuitry. In the case of SLURP, circlets with limited interactive capabilities can be implemented as SLUs; the description of the circuitry is completely device-specific, but the description of its properties is predominantly device-independent.

For general applicability of circlets, their properties must be describable in a device-independent way. One assumption made here is that circlets involve circuitry that is oriented to a two-dimensional rectangular grid, corresponding to a configurable logic array. Thus, the notion of a bounding box, expressed in terms of the units of this grid, is generally relevant. For reasons of efficiency and simplicity, making this box a rectangle is a realistic restriction. SLURP also includes placement constraints, and these are specified in a device-independent way, being suitable for any grid-oriented model. However, the existence of such constraints *per se* seems at first sight undesirable, unless it is deemed useful to make the components of the rectangular grid non-uniform in some device-independent way.

The means by which circlets can interact with their environments will be discussed in the next section. This influences the way in which the input/output properties of circlets are specified. SLURP shows one example: mapped register input/output. The description mechanism used there is nearly device-independent, one obvious generalisation being to allow horizontally-oriented registers as well as vertically-oriented.

In addition to circuitry properties already seen in the discussion of SLURP, there are two other important SLU properties included in [1] that merit inclusion as generic circlet properties:

- replication; and
- parametrisation.

Replication is a means of specifying that certain sub-circuits occur repeatedly within a circuit. Examples include bit-sliced arithmetic circuitry and systolic-array circuitry. The two main reasons for identifying replication are that circuit descriptions may be shortened, and that some technologies (e.g., the XC6200) allow configuration-time parallelism by instantiating all sub-circuit copies simultaneously. A pure form of replication description requires both a sub-box identifying the sub-circuit and also a map indicating where the replicas are to be placed; this can be done easily in a device-dependent way. However, there also seems a case for allowing the explicit specification of circuitry that is specifically associated with the collection of replicas. This would be of use for obtaining an abstract representation of the carry or cascade wiring made available in many FPGA technologies, e.g., the Altera FLEX 10K or Xilink XC4000 families.

Parametrisation is a means of specifying that certain sub-circuits within a circuit can be altered. This is a route to defining a family of related circuits that vary only in some sub-circuitry, allowing both shortened descriptions and efficiency through the use of partial reconfiguration. Description of parametrisation requires a sub-box identifying the sub-circuit that is parametrised. There are a number of possibilities for how the actual instantiated sub-circuits are described, and these depend on how the circlet interacts with its environment. The possibilities range from the circlet description including all allowable sub-circuits, to the environment being able to supply required sub-circuit descriptions. The former would be suitable for something like a bit-sliced arithmetic circuit, with variant circuitry at the least and most significant bit ends; the latter would be appropriate for applets supporting partial-evaluation style operation. The best ways to proceed will become clearer with the continuing advance of research into the subject of partial reconfiguration.

Aside from those circuitry properties already arising from the author's research into virtual circuitry and SLUs, there will be others. One obvious possibility is information about the timing properties of the circuitry. This could include a minimum clock frequency for synchronous circuitry to work; a longer-term agenda would be to provide support for a mixture of synchronous and asynchronous circuitry. An initial simplistic model is to envisage a single global clock propagated to all points on the rectangular grid.

While properties of circlet circuitry ultimately refer to their execution environment, the key independent factor is how actual circuitry is described, and a discussion of this occupies the rest of the section.

The first point to note about the circuitry representation is that it is likely to involve a form intermediate between a high-level human-oriented description and a low-level technology-oriented description. This is analogous to the use of Java applets for network computing. A high-level Java language description is compiled off-line into a byte code program for the Java Virtual Machine. This byte code program is the portable representation of the applet. The program is run by using an interpreter that emulates the Java Virtual Machine. Figure 1 shows the equivalent process when circlets are involved. The key difference is the second stage. Essentially, it is a second compilation step, to produce native circuitry for subsequent use. This is in contrast to the interpretative style of Java applets. It is possible to conceive the use of interpretation, where configurable logic implemented an interpreter for more abstract circuitry, but this is not likely to be an efficient use of either area or time resource. A benefit of the compilation style is that it is one-off, which means that it is more acceptable for this to take a non-trivial amount of computation time. The

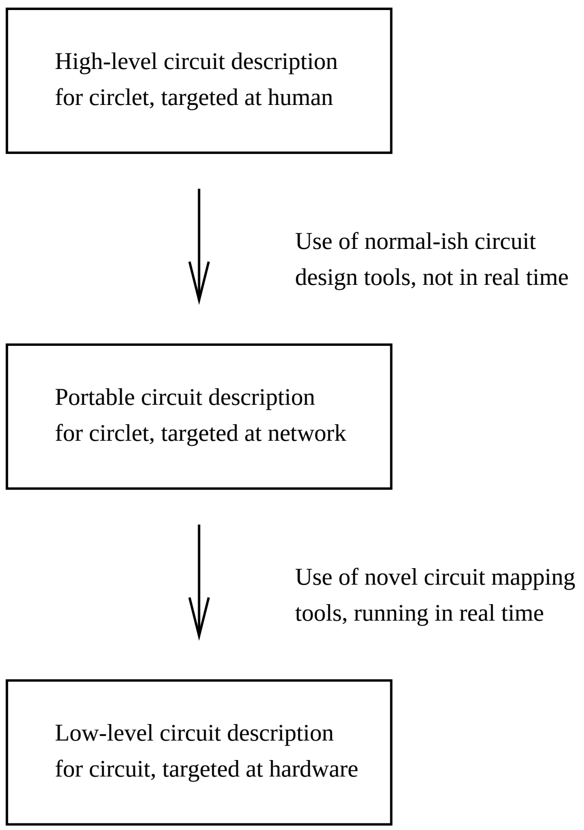


Figure 1: Compilation and execution of a circlet

major issue is the level at which the portable intermediate form is placed, to yield the best combination of flexibility and efficiency.

In general, any description of circuitry must specify a *graph*: a collection of vertices corresponding to nodes with computational capabilities, and a collection of edges corresponding to connections between nodes. However, there are many ways of specifying a graph, apart from the explicit way of just listing all of the vertices and edges.

For SLURP, the description of the graph is expressed by configuration data for the XC6200 FPGA technology. The fixed physical structure of an XC6200 consists of a collection of multiplexors with configurable select inputs, connected in standard patterns following a two-dimensional grid layout. Thus, the configuration data is a set of values for multiplexor select inputs, and so the SLURP graph description is in terms of a physical graph with fixed edges and slightly-configurable vertices.

Of course, there is more structure to the XC6200 architecture. The underlying multiplexor graph serves to implement a higher-level graph, in which vertices are one-bit function units or one-bit input/output units, arranged on a two-dimensional grid, and edges are connections between units, following the grid pattern. It would not be hard to devise a binary-encoded description for such graphs, and use this as the form for representing circuitry. In fact, the genesis of such a representation can be found in the author's SPODE circuit description system [2]; what is needed here is a variable-free SPODE description.

To give an outline of SPODE, it is centred around the assumption of a two-dimensional array of cells that corresponds directly to the logic array of a Xilinx XC6200 device. For any cell at a particular column and row position, SPODE allows the specification of:

- the function computed within the cell: any two-input boolean function or two-input multiplexor, or 'no function';
- the flip-flop type contained within the cell: D-type or constant and fed-back or not, or 'no flip-flop'; and
- internal wiring within the cell.

Of these, the function and flip-flop types are relatively device-dependent, although they actually correspond exactly to what is available in an XC6200 cell; this reflects the generality of the XC6200 as a fine-grain cell architecture. The internal cell wiring is also reasonably XC6200-specific. It encompasses wiring to and from neighbouring cells, to the function unit inputs and from the function unit or flip-flop output.

In addition to the description of cells, SPODE also includes *slices*, which are groups of neighbouring cells in columns or rows. Slices contain wiring that is global over the cells within the slice. In the case of the XC6200, use of $1 \times 4$, $4 \times 1$, $1 \times 16$ and $16 \times 1$ size slices allows description of length-4 and length-16 FastLANE wires that can fly over cells in columns or rows. Slices can be viewed as part of an area hierarchy that runs from the whole array, at the top, down through the various slice sizes, to individual cells at the bottom.

The advantage of this sort of representation, over raw configuration data, is that it is more general-purpose, suitable for any circuitry composed of fine-grain units laid out on a two-dimensional interconnection grid. However, it is sufficiently low-level that it should be a simple process to convert it into the form necessary to configure an actual technology, such as the XC6200. A key word is *should*, since unpleasant implementation details, such as the infamous 'negating multiplexor' feature of the XC6200, may make the process harder than it need be.

The previous paragraph refers to 'fine-grain units'. A significant question concerns the appropriate level of granularity for the functions of the components occupying the rectangular grid. For example, the use of only two-input logic gates might usefully be expanded to three-input or four-input gates. Going further, the single logic gate might usefully be extended to a small number of interconnected logic gates. Such extensions are tempting, in view of the fact that a majority of FPGA technologies have array cells that are significantly more beefy than the cells of the XC6200. Directly mapping a fine-grain abstract grid to these coarse-grain technologies would be extremely wasteful of cells' power.

The relative merits and demerits of fine-grain and coarse-grain FPGA architectures are still much debated and so, in the absence of consensus, it would be premature to prescribe a particular variety of logic circuit description for circlets. However, one powerful influence comes from the world of computer networking, which is concerned with bridging the differences between different computers. There, it is normal to look for lowest common denominators. For example, the Internet Protocol (IP) is pitched at the simplest type of network service imaginable, to enhance portability. As another example, the Java Virtual Machine for running Java applets comes from the RISC school of computer architecture, rather than the CISC school. This suggests that a simple two-input gate model would be the most appropriate for circlet circuit descriptions. First and foremost, it is clear that this could be directly mapped on to all physical architectures. Second, where the physical architecture is coarse-grain, there is scope for efficiency by combining a collection of neighbouring abstract cells into a single real cell. The neighbouring cells would appear as a small logic circuit, and numerous algorithms are already used by design tools for mapping circuits to technologies (e.g., [12] describes such mapping for the XC4000, which probably has the most complex cell architecture). The key is to tune these algorithms for very small circuits, so that they are practicable to use in real time.

Generalisation by adopting a higher level of abstraction, as when moving from a multiplexor circuit description to a boolean logic circuit description, seems attractive. A different alternative would be to generalise the low-level vertex type, from a simple one-bit multiplexor to a multi-bit lookup table (LUT). A practical motivation would be that LUTs, in one form or another, underpin most physical FPGA technologies, and so a LUT circuit description may be amenable to very speedy implementation. The catch is that that the differences between technologies are largely characterised by the differences in fixed interconnections between LUTs. For example, the Altera FLEX10K has one four-input LUT, the new Atmel AT40K has two three-input LUTs and the Xilinx XC4000 has two four-input and one three-input LUT. Because of this, any description using LUT circuits is likely to be very device-dependent.

An obvious question is whether further generalisation, by adopting levels of abstraction higher than the grid-oriented boolean circuit, is productive. There are two possible immediate directions: embracing more complex function unit operations, and loosening up the need for grid-oriented inter-connections. While both have the merit of reducing technology dependence, they both introduce the danger of making the cost of processing circuitry descriptions prohibitive. The mapping of complex operations to simpler logic operations may be non-trivial, and the results may be inefficient. The automatic placement and routing needed to arrange components and netlists into efficient grid-oriented patterns is well-known as a time-consuming problem, and so is not suitable as part of a dynamic configuration process. Ruling out this level of abstraction also rules out any even-higher levels, for example the use of VHDL or Viewlogic circuit descriptions.

To summarise this section, the author believes that a fine-grain abstract array architecture looks promising for portable circuit representations. The basis of the representation would be a collection of simple cells arranged on a rectangular grid, with each cell containing a two-input logic gate followed by a D-type flip-flop (a common global clock signal is made available to each flip-flop). A purist wiring regime would allow only communication to the four column and row neighbours. In fact, aside from the inclusion of flip-flops, such an architecture is somewhat akin to that of the Algotronix CAL1024, ancestor of the XC6200. In the longer term, additional wiring capability seems necessary. One interesting line would be inspired by the new Acmel AT40K, which includes wiring between diagonally-neighbouring cells. However, more mainstreamly, a natural extension would be to include additional wiring planes, each allowing longer-distance wiring along columns and rows.

## 4 Execution environment for circlets

One of the best features of using Java applets is that their environment includes a range of APIs that allow applets to influence their execution environment in very high-level, and relatively safe, ways. For circlets, a more modest, but realistic, aim is that they can influence their environment in rather low-level, but still safe, ways. There are two general components to this influence: one is interaction with other SLUs, and the other is interaction with other system hardware and software. The author believes that the discussion of these matters for SLUs in general, contained in [3], is extremely relevant to circlets in particular. In fact, so much so that there is little need to revisit the issues here, except to stress some particularly important points.

First, the matter of interaction between SLUs centres upon issues of placement and routing; that is, where SLUs are placed and how information is communicated between SLUs, rather than the massively-popular issue of how best to place and route circuits. The intention is that SLU placement and routing must be very efficient to carry out dynamically, so the problem has to be constrained appropriately. For SLU-based virtual circuitry, placement follows one of two models:

- the sea of accelerators, where the position of one SLU is not influenced by the positions of other SLUs (except that it cannot overlap!); and
- the parallel harness, where the positions of a collection of SLUs are regularly placed in a predefined simple architecture.

There has been a significant amount of research into placement under the sea of accelerators regime, both by the author [1] and, outwith the virtual circuitry context, by Diessel and ElGindy [8].

Routing between SLUs is discussed in a recent paper by the author and Adam Donlin [5]. This develops three models, placed along a continuum of possibilities:

1. parallel communication achieved by juxtaposition of SLUs or by a fixed wiring graph (e.g., a tree or a torus) between SLUs;
2. parallel or sequential communication achieved by connecting SLUs to a reconfigurable switch fabric (e.g., a crossbar or a bus); and
3. sequential communication achieved by directing data flow between SLUs using a URISC (Ultimate RISC) processor core — URISC has a single instruction type: *move*.

In all three models, the virtual circuitry operating system supplies some fixed configured circuitry: wiring in Model 1 (unless juxtaposition is used exclusively); switch fabric and wiring in Model 2; and the URISC core in Model 3. Model 2 has parallel and sequential variants, and so is positioned at a transition point between parallel Model 1 and sequential Model 3. It involves a trade-off between the ability to support parallelism and both the time required for reconfiguration and the area required for the switch fabric.

An interesting feature of these models is that they introduce situations where routing is the primary object of reconfiguration, elevating it from its traditional role as a second class design object. This in itself highlights some interesting deficiencies in current design capture techniques and design tools. For example, embedding routing information into VHDL requires the *ad hoc* use of the `attribute` mechanism. VHDL is more concerned with a detailed specification of functionality coupled to a fairly abstract specification of the interconnection of that functionality. For the run-time reconfigurable designs discussed here, a means of describing the routing layout solely in a concrete manner is necessary. Low level HDL systems like SPODE [2] and JERC [9] show more promise in this area. Visual techniques, such as schematic capture, are potentially useful, but must become flexible enough to allow the specification of 'disconnected' routes, and to do so at a technology-specific level. Heavy emphasis on design synthesis has resulted in tools and techniques that are suitable for crafting abstract circuit descriptions. The mixture of abstract and more technology-specific descriptions necessary for SLU-based reconfigurable designs is less well served.

The second matter, of interaction between SLUs and their hardware and software environment, is extensively discussed in [3], and does not need to be covered again here. However, there are two specific points worthy of note. The first is that the hardware environment may include other on-chip components; some existing FPGA technologies include extra RAM components, and the situation will become increasingly complicated with the rise of system-level integration. Future research will investigate whether the presence of such other components needs to be accounted for within circlet descriptions.

The second point is how a circlet should be viewed in the context of an environment for supporting Java-style applets, since an obviously-desirable combination for a network computing system would involve both circlets and Java applets. The immediate answer is that the functionality of a circlet can be viewed as being the implementation of a method contained within a Java object. One way of doing this would be to add a new API designed to allow interaction with circlets in general; such a solution would be straightforward to implement and, as a mechanism, would have a lot in common with existing mechanisms, such as the use of Hardware Objects [7]. A better longer-term solution would be to adjust the Java run-time system so that it is capable of dynamically choosing between function implementations expressed in circlet form, byte code form and native code form.

## 5 Conclusions and Future Work

This paper has reported the first stages of research into a 'killer application' of configurable logic technology: the exploitation of the inherent parallelism and fast execution of flexible circuitry in a network computing environment. The SLURP demonstration system illustrates most of the logical concepts necessary for a circlet approach to network computing. Although the actual circuitry is described in a device-dependent form for SLURP, this is a matter that can be improved as a matter independent of the relatively device-independent framework.

Many of the concepts related to circlets have already been uncovered in the context of earlier work on virtual circuitry; in essence, a circlet is a particular case of a Swappable Logic Unit — the major new component is the need for portability of circlet descriptions. The new work enriches the virtual circuitry work, both by acting as a powerful application and by suggesting further avenues for virtual circuitry advancement.

Consideration of appropriate intermediate forms of portable circuit description has benefited from other earlier work on lowish-level circuit specification methods. This work continues, through a current project to merge the facilities of the author's SPODE [2] and Steven Guccione's JERC [9]. Not only will the combined system be available as a Java-based lowish-level design system with a graphical user interface, but it is also available to act as a lowish-level technology mapping library for higher-level design tools.

Overall, the comination of circlet work, virtual circuitry work and SPODE/JERC work points to a need for a more flexible view of configurable logic design tools. Conventional tools follow the model of normal hardware design tools, with FPGA technology viewed as a final-stage alternative to VLSI. This ignores the key feature of FPGA technology, particularly as it develops and matures: it is a soft technology, not a hard technology. There has to be a blend of ideas from the hardware and software worlds.

Further research will continue into the most desirable abstract form for portable circlet descriptions. The initial thinking is that a fine-grain cell model will be more promising. However, future work will investigate this issue properly, to compare the advantages and disadvantages of mapping finer-grain models to coarser-grain technologies, and mapping coarser-grain models to finer-grain technologies. This will take place against a background of rapid technological evolution, not only of configurable logic technology but also of configurable logic as just one component of systems on single chips. The author profoundly hopes that such evolution will take into account the requirements posed by virtual circuitry and circlet models. At the moment, technological advancement seems largely geared to delivering improvements to system designers following a very conventional hardware design approach.

Of course, the proof of concepts such as circlets is in demonstrating benefits for real applications. In the case of the author, these applications centre around the use of 'active' communication protocols, where executable components of protocol engines are transmitted over networks; such components may be in program form or in circlet form. Currently, active protocols are being investigated for use with a very high-speed optical local area network [6], and this includes the use of both circlets and also locally-based virtual circuitry.

## 6 Acknowledgements

The author acknowledges the continuing support of the Xilinx Development Corporation and the Virtual Computer Corporation. Thanks are also due to Simon Hatliffe for his implementation work on the SLURP system.

## References

[1] Brebner, "A Virtual Hardware Operating System for the Xilinx XC6200", Proc. 6th International Workshop on Field-Programmable Logic and Applications, Springer LNCS 1142, 1996, pp.327–336.

[2] Brebner, "CHASTE: a Hardware-Software Codesign Testbed for the Xilinx XC6200", Proc. 4th

Reconfigurable Architecture Workshop, ITpress Verlag, 1997, pp.16–23.

[3] Brebner, "The Swappable Logic Unit: a Paradigm for Virtual Hardware", Proc. 5th Annual IEEE Symposium on Custom Computing Machines, IEEE Computer Society Press 1997, pp.77–86.

[4] Brebner, "Automatic Identification of Swappable Logic Units in XC6200 Circuitry", Proc. 7th International Workshop on Field-Programmable Logic and Applications, Springer LNCS 1304, 1997, pp.173–182.

[5] Brebner and Donlin, "Runtime Reconfigurable Routing", to appear in Proc. 5th Reconfigurable Architecture Workshop, Orlando, March 1997.

[6] Brebner and Pooley, "ECOLE: A Configurable Environment for a Local Optical Network of Workstations", to appear in Proc. 2nd Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing, Las Vegas, January 1997.

[7] Casselman, Thornburg and Schewel, "Creation of Hardware Objects in a Reconfigurable Computer", Proc. 5th International Workshop on Field-Programmable Logic and Applications, Springer LNCS 975, 1995, pp.119–1128.

[8] Diessel and ElGindy, "Run-Time Compaction of FPGA Designs", Proc. 7th International Workshop on Field-Programmable Logic and Applications, Springer LNCS 1304, 1997, pp.131–140.

[9] Lechner and Guccione, "The Java environment for reconfigurable computing", Proc. 7th International Workshop on Field Programmable Logic and Applications, Springer LNCS 1304, 1997, pp.284–293.

[10] Kean, Churcher and Wilkie, "XC6200 fastmap processor interface," Proc. 5th International Workshop on Field Programmable Logic and Applications, Springer LNCS 975, 1995, pp.36–43.

[11] Negroponte, *Being Digital*, New York:Knopf 1995.

[12] Inuani and Saul, "Technology Mapping of Heterogeneous LUT-based FPGAs", Proc. 7th International Workshop on Field-Programmable Logic and Applications, Springer LNCS 1304, 1997, pp.223–234.

[13] Shand, "A Case Study of Algorithm Implementation in Reconfigurable Hardware and Software", Proc. 7th International Workshop on Field-Programmable Logic and Applications, Springer LNCS 1304, 1997, pp.333–343.

[14] Xilinx Inc., "XC6200 Field Programmable Gate Arrays" (version 1.10), Xilinx 1997.