Runtime Reconfigurable Routing

Gordon Brebner and Adam Donlin

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

Abstract. This paper is concerned with practicable solutions to a dynamic circuit reconfiguration problem: how to perform runtime routing of data between blocks of circuitry. The solutions use a 'virtual circuitry' approach based on the notion of Swappable Logic Units (SLUs). They involve a continuum of three types of routing model in which communication channels are made available using some form of extra configured logic supplied by an operating system. These models involve trade-offs between flexibility, speed and cell count; however, all stop short of any impractical attempt at arbitrary routing at run time. The models also illustrate a blurring of traditional notions of 'hardware' and 'software', at a point where circuitry meets instruction sequences.

1 Introduction

Placement and routing are demanding parts of the normal design process for configurable logic circuitry. They operate on timescales of minutes, even hours, which are in stark contrast to timescales for circuit reconfiguration of microseconds or milliseconds. This appears to deal a fatal blow to any hopes of implementing reconfigurable circuitry that changes rapidly and frequently at run time, since the necessary placement and routing cannot be performed at anything like the necessary rate.

However, there are some partial solutions to the problem that do not achieve full generality, but do allow practical systems to be built. One approach is to operate with circuit descriptions at a very low level — a technology-specific array of configurable logic cells — and write programs that manage cell contents and wiring between cells, as a special-case alternative to higher-level placement and routing. Another approach, which has been used in most 'virtual hardware' style work, is to have a collection of pre-placed and routed configurations, and then load these configurations in a scheduled sequence over time in an application-specific manner.

This paper is concerned with a more general technique for partially solving the dynamic circuit reconfiguration problem, namely the 'virtual hardware operating system' approach based on the notion of Swappable Logic Units (SLUs) — virtual hardware analogues of pages or segments in virtual memory [3, 4]. At this point, the authors advocate a change in terminology: to use the phrase 'virtual circuitry' instead of 'virtual hardware'. Virtual circuitry concerns with programmed circuitry — in contrast to a conventional view of software, restricted to a concern with programmed instruction sequences.

The subject of how SLUs can be placed on a configurable array at run time has been addressed in earlier work [3, 5], and research is continuing on this, to explore alternative and better algorithms. Here, the concern is with routing: how communication channels between SLUs can be provided at run time. The initial work on SLUs has involved a 'sea of accelerators' model, where there are *no* communication channels (although an external processor could act as an agent for transferring data between SLUs). The work reported here involves a continuum of three types of routing model in which communication channels are made available using some form of extra configured logic supplied by the operating system. These involve trade-offs between flexibility, speed and cell count; however, all stop short of any attempt at arbitrary routing at run time, since this is deemed impractical. The models also illustrate a blurring of traditional notions of hardware and software, at a point where circuitry meets instruction sequences.

The three models are all implemented, or being implemented, using the Xilinx XC6200 FPGA technology; however, the underpinning ideas are applicable to most available technologies. The models are:

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*.

More detailed descriptions and discussions of each model are contained in Sections 2, 3 and 4 respectively. However, some general discussion of the relative characteristics of the models is appropriate at this point.

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. Note that, for Model 3, the wiring to SLUs is the physical wiring provided on chip for memory-style reads and writes to array cells, and so is not configured by the operating system. The extra support circuitry, together with the circuitry for the SLUs, forms the coarse grain reconfigurability supported by the operating system. It delivers a basic circuit architecture in which there are processing elements together with a communication mechanism.

A finer grain of reconfigurability is available for the SLUs themselves, using 'parametrised areas', as described in [3], which are a means of supporting the notion of partial evaluation. However, finer grain reconfigurability is also available in Models 2 and 3, since the communication mechanism can be programmed. In the case of Model 2, the programming information is in the form of circuit configuration data for the switch fabric; in the case of Model 3, it is in the form of a move-instruction program for the URISC processor core. This illuminates the 'hardware-software convergence'. In fact, the URISC core can be viewed as being a switching fabric — the execution of a move instruction corresponds to the instantaneous creation of a wiring path between two SLUs. This represents the ultimate in dynamic reconfigurable routing: a connection that only lasts for as long as a single data transfer.

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. Model 3 does not allow parallel communications between SLUs; the processor acts as a sequencing unit, and the URISC program determines the order in which communication takes place between SLUs. Thus, at first sight, Model 3 might seem distinctly worse than the others; however, it does have some advantages. One is that the URISC core can also be used for other purposes, such as memory accessing and indeed actual array configuration. Another is that Model 3 makes use of existing physical wiring, which is faster than configured wiring and also saves the cell overhead of wiring to a switch fabric. Finally, Model 3 gives complete flexibility, by allowing communication between any two SLUs, anywhere on the logic array. In essence, Model 3 is the same as the 'sea of accelerators' plus processor as a communications agent, but with the processor being dynamically configured on the logic array.

2 Fixed parallel wiring fabrics

Fixed parallel wiring fabrics correspond to Model 1 of Section 1. These are suitable for use in cases where the required architecture consists of a collection of parallel processing elements (PEs) with a fixed, regular interconnection pattern. This includes things like systolic pipelined architectures, as well as more generic parallel processing arrays. The expectation is one of highly-parallel communication between PEs, with input and output occurring at the perimeter of the architecture, not directly to and from individual PEs. In order to keep placement of SLUs by the operating system tractible, a general requirement is that they are rectangular. A further requirement for this model is that all of the SLUs are the same size; indeed, they might carry out the same PE function, for example in a systolic array.

The simplest form of wiring fabric for the operating system is one in which there is no explicit wiring (clearly, there are also no cells wasted on configured wiring). In such cases, communication can take place between SLUs that are juxtaposed in the array, since their interfaces will abut. If this approach is used, then it is desirable that SLUs are placed in a way that ensures a dense tiling of the configurable logic areas, to avoid wastage of cells. The possibilities are familiar from the study of similar-style VLSI architectures [9]: dense one-dimensional arrays are possible using triangular or rectangular components; dense two-dimensional arrays are possible using triangular, rectangular or hexagonal components. Here, the components (SLUs) must be rectangular, which restricts things rather. However, the rectangular tilings can be joined by a pseudo-hexagonal tiling using offset rectangular components. The extra tiling permits communication with six neighbours, rather than only four.

For interconnection graphs other than the simple meshes possible using juxtaposition, care is needed on several grounds. The additional wiring must not consume too much cell area, and also must not require the operating system to spend more than a non-trivial amount of time on computing configuration data for the wiring or on actually configuring the wiring. Two possibilities are the binary tree and the torus. The H-tree layout for binary trees is well-known from the world of VLSI. Note that, if SLUs are allowed to differ in size and, in particular, if sizes decrease from root to leaves in the tree, then a layout using juxtaposition only is possible, as illustrated by a special-case design in [2].

A torus architecture extends either the one-dimensional or two-dimensional rectangular grid architectures so that the endpoint PEs of a row or column can communicate directly. There are two simple ways of laying out a two-dimensional torus. The first layout involves just the addition of single wires from one end to another; disadvantages are the cell resource consumed, the delays in communicating over the long wire and the overall asymmetry introduced. The other layout involves revised placement, in which the SLUs are laid out in a ring formation, alleviating some of the problems of this simple layout. In particular, all wires are four or eight cells long, independently of the side length of the torus. This particular approach is derived from a long-standing VLSI layout for toroidally-connected parallel architectures [1].

At this time, it does not seem useful to investigate more complicated wiring fabrics, since they are unlikely to be efficient for fast reconfiguration or in terms of area usage. In [1], there are fairly optimal VLSI layouts for several popular parallel interconnection graphs, e.g., hypercubes, de Bruijn graphs and cube-connected cycles, and the rather complex details of these layouts serve to confirm this assessment.

3 Configurable switching fabrics

The regular static interconnection of SLUs was considered in Section 2; more general interconnections are considered in this section. The main conceptual difference is that the communication pattern of SLUs in the model of Section 2 is known prior to configuration time, allowing abutment, tiling and hard instantiation of specific routing between abutted SLUs to satisfy interconnection requirements. For the model of this section, however, the exact communication pattern is not known in advance of configuration, so flexible interconnection is made available through run time setting of switches in a switching fabric that has been configured on the array.

The aim is to reconfigure the minimum amount of routing necessary to establish and break interconnections between SLUs. The switching fabric, and connections to it, are configured on the array, but routes are left uncompleted at key switching points. For example, the structure could be that of a butterfly or other parallel interconnection network. Partial run-time reconfiguration is used to apply the minimal reconfigurations required at switching points. By this method, the amount of reconfiguration required to interconnect SLU ports is minimised. Computational overheads for the switch setting are also reduced by the constraints imposed through the incomplete routing harness.

In the case of a crossbar switching fabric, this ground has already been explored by others. For example, Eggers *et al* [7] describe a design containing a single reconfigurable crossbar switch. The switch settings are known in advance, being dependent on the particular application, so the configuration data can be pre-computed. This allows the 32×32 crossbar, implemented using Atmel AT6000 technology, to be reconfigured within around 700 ns. In the general operating system context here, this proof of concept can be used to underpin a mechanism where a crossbar switch is used to interconnect a collection of SLUs.

Parallel switching fabrics, such as butterflies or crossbars, are not particularly feasible, given the densities of current FPGA technologies, because they require cell areas and configuration times that are supra-linear in the number of connected SLUs. The situation can be improved by trading away the parallelism in favour of sequentiality, and so simplifying the switching fabric. One example is the use of a bus as a switching fabric; arbitration of the bus enforces sequential communication. The reconfigurable part of the switching fabric is at the points where the connections from SLUs reach the bus. At any time, only two connections are joined to the bus: these determine the communicating SLUs. The switch setting is changed by first disabling the two current connections, and then enabling the two new connections. This is very efficient in terms of reconfiguration time.

The transition from parallel to sequential switching fabrics marks a transition that is most often viewed as a distinction between hardware — with inherently parallel circuitry — and software — with inherently sequential programs. However, the transition here is, in fact, in the world of programmable circuitry, poised somewhere between the hardware and software traditions. Note that, unlike the circuitry described in Section 2, the circuitry of this section involves two levels of configuration: an upper level, less frequent, for configuring a switching fabric and a lower level, more frequent, for setting particular switching patterns.

4 Configurable sequential data flow fabrics

The Ultimate RISC (URISC) [8] is a minimal processor architecture with only one instruction: move memory to memory. Devices that are normally elements of a standard processor core are on the system bus, and are included in the memory map of the URISC core, allowing operands and results to be moved to and from the device. For example, the URISC core contains no general purpose ALU — instead, the ALU resides on the system bus and is memory addressable. Performing addition is then a matter of instructing the URISC core to move the addition operands to the memory locations corresponding to the adder input registers and to read results from the ALU's memory mapped accumulator. Thus, the URISC core is acting as a communications agent, actively transferring data from point to point across the system bus. This is reminiscent of the sequential switching fabrics discussed in Section 3. The datapath and control requirements of the URISC core are lean and simple, making the implementation of the URISC core on a configurable logic array feasible.

Two points are worth noting about this implementation. First, there are essentially no static system bus devices. Instead, SLUs are dynamically placed and replaced on the configurable array alongside the URISC core itself. The set of SLUs resident on the array correspond to the devices currently present on the URISC system bus. Second, there need be no direct implementation of the URISC system bus on the cell array. Instead, the URISC core can exploit a memory style co-processor interface to the logic implemented on the configurable array, such as the XC6200 FastMap interface. Rather than passing data over the array's routing resources, the URISC core can exploit the random access interface to both read from and write to registers within source and destination SLUs. A particular advantage of implementing the URISC core on the cell array is that all movement of data occurs on chip.

Control for the system comes from the operating system, which specifies a series of URISC move instructions – a URISC program defining the desired motion of data from SLU to SLU across the system bus. The URISC program's move instructions may be simply an explicit set of transfers between SLUs. Alternatively, since the URISC machine is capable of computation, the order of communication may be defined *algorithmically* and then realised as a URISC program. In this way, physical routing can be dynamically defined by the execution of a software algorithm.

Of course, the communication implemented through the URISC is inherently serialised. The architecture of the basic URISC machine and the configurable array both have a rôle to play in this. The URISC is a sequential machine, executing single `move` instructions in a strictly defined sequence. Architectural modifications to that basic architecture are conceivable, but must be carefully balanced against any additional resource utilisation.

Modifications to the basic memory-style co-processor interface, that could benefit the standard URISC machine, are conceivable. The ability to use 'wildcards' within the address specification — as is possible with the XC6200 architecture — would allow multicast communication between SLUs. Additionally, array accesses through the memory style interface of the XC6200, for example, are column based and exploit a masking feature to identify the particular sections of an array column that are to be read or written. SLU communications not requiring the full width of a co-processor interface transfer could be piggy backed within a single transfer and, using column masking facilities, delivered to the respective destination SLUs. This, of course, requires some consideration in the dynamic placement of SLUs to ensure that input/output ports are appropriately aligned. Certain application classes, for example, systolic, exhibit the regularity of layout that can be exploited.

The serialisation of communication may imply a drop in the overall throughput and performance of the system. However, this is not necessarily the case. The memory style co-processor interface is a physical wiring resource, and may be exploited faster than a configured route on the cell array. A memory style interface also has a single defined delay for arbitrary accesses to the cell array, as opposed to configured routing where the effective delay is equivalent to the longest delay on the routes used within the connection. Since the URISC is capitalising on an existing physical resource, there is no need to dedicate additional areas of the cell array to routing tracks between SLUs. Array utilisation is held constant at those resources required to implement the URISC core, as opposed to a varying level of utilisation for configured routing based on the particular width and density of interconnect required. For SLUs requiring wide bit-parallel interconnections, configured routing would consume significant array resources, whereas the URISC model readily supports bit-parallel communication with no such additional resource consumption.

5 Conclusions

This paper has presented a discussion of a continuum of reconfigurable routing, spanning from reconfigurable models that support highly parallel communication to models which support intrinsically serialised communication. Three significant points within that continuum have been identified, and the reconfigurable models they represent discussed. Detailed performance analyses have been omitted from this paper, but will be provided through further work.

One particularly interesting conclusion concerns the benefits of the serialising effect of configurable switch and sequential dataflow models for arbitrating between SLUs that access contested resources. Consider two SLUs that desire simultaneous access to a particular cell array resource which cannot be shared. Arbitration is clearly required and, by applying standard virtual circuitry, may be achieved automatically through the swapping of SLUs on and off the cell array – the act of swapping itself serialises access of the SLUs to the contested resource. If, however, SLUs require a tightly interleaved access schedule, automatic arbitration cannot be gained by the application of standard virtual circuitry: the system would be overwhelmed by the penalty of frequently reconfiguring the array to swap SLUs.

Instead, provided a space-performance tradeoff can be justified to allow both SLUs to be simultaneously resident on *some* part of the cell array, the configurable switch and configurable sequential dataflow models of reconfigurable routing may be used to support a tightly interleaved access schedule. Both of these models exhibit levels of sequentiality and serialisation of communications which, when either model is used to interconnect the SLUs to the contested resource, automatically provides the necessary arbitration. This, combined with the reduced reconfiguration costs characteristic of both models, allows tighter interleavings to be supported than previously capable through standard virtual circuitry. In conclusion, the serialisation imposed by the reconfigurable routing models can indeed be used to particular advantage and, in combination with certain reconfigurable routing models, actually increases the applicability of virtual circuitry in general.

References

1. Brebner, "Relating Routing Graphs and Two-dimensional Grids", Proc. VLSI : Algorithms and Architectures, North Holland Publ., 1984, pp.221–231.
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 LNCS 975, 1995, pp.429–438.
3. 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.
4. 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.
5. Burns, Donlin, Hogg, Singh and de Wit, "A Dynamic Reconfiguration Run Time System", Proc. 5th Annual IEEE Symposium on Custom Computing Machines, IEEE Computer Society Press 1997, pp.66–75.
6. Donlin, "A Dynamically Self-Modifying Processor Architecture and its Application to Active Networking", Working Paper, Department of Computer Science, University of Edinburgh, September 1997.
7. Eggers, Lysaght, Dick and McGregor, "Fast Reconfigurable Crossbar Switching in FPGAs", Proc. 6th International Workshop on Field-Programmable Logic and Applications, Springer LNCS 1142, 1996, pp.297–306.
8. Jones, "The Ultimate RISC", Computer Architecture News **16** 3, June 1988, pp.48–55.
9. Mead and Conway, *Introduction to VLSI Systems*, Reading:Addison-Wesley 1980.