

A Dynamic Reconfiguration Run-Time System¹

Jim Burns, Adam Donlin, Jonathan Hogg, Satnam Singh, Mark de Wit
The Department of Computing Science
The University of Glasgow
Glasgow G12 8RZ, United Kingdom
{jim,adam,jonathan,satnam,dew}@dcs.gla.ac.uk

Abstract

The feasibility of run-time reconfiguration of FPGAs has been established by a large number of case studies. However, these systems have typically involved an ad hoc combination of hardware and software. The software that manages the dynamic reconfiguration is typically specialised to one application and one hardware configuration. We present three different applications of dynamic reconfiguration, based on research activities at Glasgow University, and extract a set of common requirements. We present the design of an extensible run-time system for managing the dynamic reconfiguration of FPGAs, motivated by these requirements. The system is called RAGE, and incorporates operating-system style services that permit sophisticated and high level operations on circuits.

1 Introduction

Dynamic reconfiguration of FPGAs has recently become viable with the introduction of devices that allow high speed partial reconfiguration, e.g., the Xilinx XC6200 series [14]. Dynamic reconfiguration is usually performed by a software system that decides when to reprogram part of the FPGA and with what. The simplest kind of run-time software simply selects a precompiled circuit and transmits the programming data directly to the FPGA.

At the University of Glasgow's Department of Computing Science, the Reconfigurable Architecture Group (RAGE) has a number of projects examining applications of dynamic reconfiguration. These applications have highlighted the need for a more complex run-time system. Rather than developing different run-time systems for each application, as is common, we have extracted a set of common requirements from several applications. This has formed the basis of the design of a proposed, core, run-time system, able to support all of these applications. There are many parallels that can be drawn between this design and conventional operating system design—as the techniques used to manage conventional resources, such as memory and the CPU, are also applicable to the management of FPGAs.

The next section of this paper describes three applications of dynamic reconfiguration and their requirements, and draws from these a set of core requirements. We follow this with an overview of the proposed system, and more detailed discussions of the rôles of the identified system components. We close with a discussion of how our system might support other applications in the field.

2 Case Studies

2.1 The FPGA/PS Project

The FPGA/PS project aims to exploit the dynamic reconfigurability of FPGAs to accelerate PostScript® rendering. The bulk of PostScript® rendering is still performed on a host computer, but the time consuming rendering of graphics for high quality printing (thousands of dots per inch) is accelerated by circuits realised on an Xilinx XC6200-based FPGA sub-system. When a different rendering operation is required (e.g. line or arc rendering), it is dynamically swapped onto the FPGA. We call this technique of swapping circuits on and off FPGAs 'virtual hardware' [8]. Although this technology is being developed to support PostScript® rendering, the general architecture is of relevance to a much wider class of applications.



Figure 1 Xilinx XC6216 PCI Board

1. This research is supported in part by UK EPSRC grant number GR/K82055

A collection of pre-routed and placed circuits reside in the main memory of the host system, ready for rapid download onto the FPGA over a PCI bus. The FPGA is housed on the board shown in Figure 1. At pre-defined intervals during PostScript® rendering, data and operation codes are passed as parameters to the virtual hardware run-time system. The system then swaps an appropriate circuit onto the FPGA and also preloads circuit input registers with values passed from the application.

As results are generated by the circuit, the application is interrupted allowing it to fetch the relevant data from some known memory location. Only one working circuit is resident on the chip at any time. Unlike other similar reconfigurable systems, our proposed system aims to factor out such common run-time support operations rather than hard-coding them into the application.

To ease memory mapping between the FPGA, the RAGE system, and the FPGA/PS application, circuits can be organised in the following ways:

1. Some circuitry is held static on the FPGA and serves as input and output ports for the dynamic circuitry. For instance, a circle rendering circuit needs (x, y) coordinates for the centre and a radius value. These static input registers would be reused when the next rendering circuit is swapped in. This policy reduces the number of cells to be configured, but makes circuit placing less flexible.
2. The input and output ports of the circuit representation are tied to the working circuit itself, and the circuit is swapped onto a pre-determined location on the FPGA.
3. The circuits are able to be placed anywhere (within known boundaries) and the RAGE system organises the mapping of circuit registers into host memory transparently. To the application, these registers are at a well known address.

Circuits which need no run-time place and route can be represented very simply, as address/data pairs which form the FPGA programming information. There is no need to have an internal representation of the FPGA itself, because no place and route need take place.

If, however, more complex swapping is required (cases 2 and 3 above) some run-time place and route is required. A full representation of all the muxes and configuration memory on the chip is maintained by the run-time system. The internal representation of circuits needs to be richer too, encompassing knowledge of ports, hierarchical circuit blocks, etc.

The circuit swapper has access to a library of circuits, and passes the programming data to the device driver. When a circuit is being swapped, it is important that the behaviour of the system is predictable. The circuit swapping mechanism should either isolate the on-chip static circuitry, or be aware that spurious results may be produced whilst swapping is in progress.

2.2 Partial Evaluation

Partial evaluation provides a methodology for reducing the resource requirements of, possibly otherwise unimplementable, circuits by utilising the nature of some circuit inputs to remain static for long periods of time relative to the other inputs. An example of this is a circuit which multiplies a stream of numbers by a constant—a situation often found in digital signal processing applications. In this example, the constant value would normally be given as an input to the circuit because, while it is a constant number for a particular stream, it may change for another. However, this requires the number to be routed through the circuit and a general multiplier to be implemented. By specialising the multiplier for a particular number, the routing resources required to feed in that constant are freed and components with now static functionality can be simplified or removed [10].

Partial evaluation is already a common technique in software compilation and execution, particularly in the field of functional programming. Here, the source text is analysed for static expressions which can be evaluated at compile-time. However, the traditional flow from hardware description to implementation involves going through hardware compilers and a generally lengthy place-and-route phase where the gates of the compiled hardware description are arranged according to the available FPGA resources. It is not feasible to do such work at run-time when a constant value is changed. As much as possible, the work of generating specialised circuits should be moved to the initial compile-time rather than run-time. This can be achieved by symbolic partial evaluation, where inputs are tagged as static, but their values are not defined. These inputs are traced through the circuit to see which components of the circuit are functionally dependant on them. Case analysis of these components can then be used to determine the different possibilities for different inputs [4].

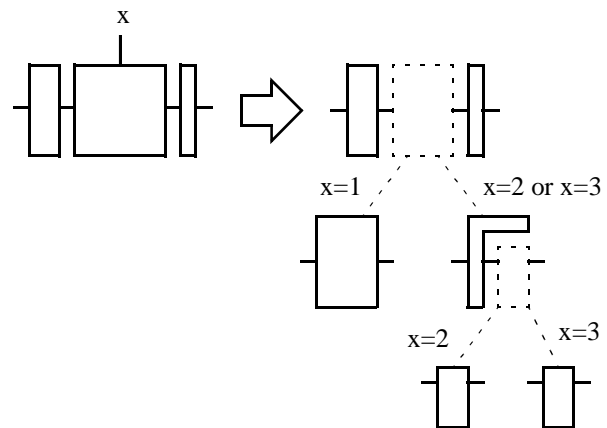


Figure 2 Symbolic partial evaluation by case analysis

This results in a tree of possible circuit components as shown in Figure 2. For any particular values of the partially

static inputs, the appropriate leaves of this tree will yield the components of a specialised circuit for those values. In order to be able to quickly build specialised circuits at run-time, the components at the leaves of the tree should be pre-placed-and-routed. This allows the components to be made resident in the appropriate combination, requiring only routing between components to be done at run-time.

Generating such a tree of components is not an easy task and represents on-going research in the group. Different branches of the tree must be inter-changeable for each other and thus must have similar FPGA footprints. This is achieved by manipulating designs given in the Ruby hardware description language [11]. Specifications in Ruby encode the layout as well as the behaviour of the circuit. The symbolic partial evaluation process transforms these specifications such that the areas of reconfiguration are localised, and the overall layout is retained as much as possible. These transformed specification trees must be placed-and-routed to produce a tree of FPGA-level components which have controlled footprints allowing them to be interchanged. Unfortunately, no automatic tool exists that is capable of doing such directed place-and-route. However, existing tools capable of doing partial place-and-routes can be utilised by judicious use of placement constraints.

2.3 Reconfigurability by Design

For circuits which are inherently reconfigurable, a large proportion of the physical design is functionally static. A few *well defined* regions of the design, however, are identified as being reconfigurable. By effecting changes solely in these areas, the functionality of the overall circuit is modified within some restricted overall area of operation. For example, a multiplier does not turn into a VRAM circuit, but a *3 multiplier may become a *7 multiplier. This effectively constitutes fine grain inherent reconfigurability at gate level as opposed to large scale reconfigurations used in the FPGA/PS project.

An important requirement in the context of fine grain reconfiguration is to have a high level representation of the circuit that in itself identifies areas of reconfigurability symbolically. This representation is submitted to the system and the corresponding circuit is made resident on the board. Later, we refer to the reconfigurable areas of the circuit by their symbolic name. By passing this symbolic reference to the system with the circuit with which it is to be substituted, we can effect the desired style of reconfiguration.

In detailed terms, the circuit representation passed to the system must therefore be at least a CAL file (a file containing a circuit representation at a low-level/device programming interface level) and some higher level symbolic representation of the circuit itself. A layer generated/imposed by the system provides a mapping between these representations—effectively fusing them together. Furthermore, the mapping must be invertible between representations. The circuit will

most likely be modified by the system upon submission. We recognise that dataflow in the system may not entirely flow downwards. The system must also be able to direct upflowing results and configuration data to the appropriate application entity. To allow the enforcement of more complex application level reasoning the application may acquire knowledge of the transformed state of its submitted circuit. A circuit may be transformed and made resident, but in its modified form would fail to fulfil application specific criteria. The detailed judgement of such criteria is beyond the scope of our proposed system, but may be suitably placed at application levels.

An optimisation to the submission of both CAL and symbolic representations is that the application submit only a symbolic representation of a pre-placed and routed circuit and allow the system to generate the programming stream level circuit representation. It is known that stream generation is a relatively simple task and can be performed rapidly. The main benefit is that the complexity of the application interface is reduced from the users point of view.

2.3.1 Example: Systolic FIR

As an example of a design which has real world practical application, we consider an n -tap systolic implementation of a finite impulse response (FIR) filter. The basic structure of a systolic FIR, shown in Figure 3, is a series of processing elements interconnected in a regular, linear fashion [6,7]. The systolic FIR we consider utilises Kean's inherently reconfigurable multiplier design [5].

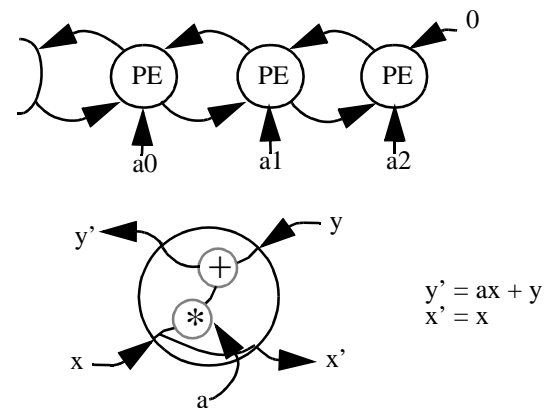


Figure 3 Systolic Array Structure

The basis behind Kean's multiplier relies on the ability to split an m -bit multiplication (with a $2m$ bit result) into two $\frac{m}{2}$ -bit multiplications. These subsequent multiplications are realised by two independent look up tables (LUTs) which are each configured to produce the multiplied value of the free coefficient by the upper and lower halves of the constant coefficient respectively. These two partial results are recon-

ciled by a 16 bit adder circuit to produce the final $2m$ -bit result.

The lookup tables are arranged to make them easy to dynamically reconfigure. Every cell in columns 0, 2, 4 and 6 is a 2 input to one output gate which encodes four columns of the truth table of a single-bit multiplier. At run-time the host makes a calculation to decide for a given 12-bit coefficient what the values of these gates should be. There are 48 cells for each 4-bit by 12-bit multiplier, giving a total of 96 cells to be reconfigured. The functionality of an individual cell is defined by a single byte. As four such elements may be passed in a single configuration cycle, the total number of cycles required, in the worst case scenario, is 24. Given a 33MHz programming interface, an entire LUT could be completely reconfigured in $1.45 \mu\text{s}$ including setup costs.

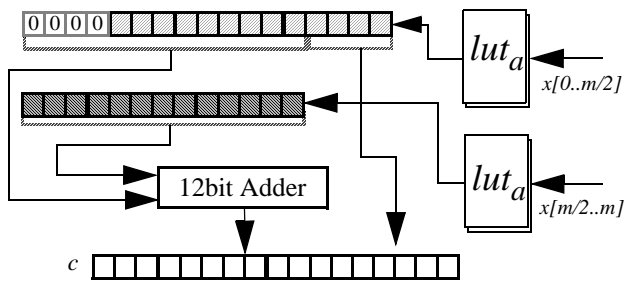


Figure 4 Reconfigurable multiplier dataflow for $a*x=c$

The systolic array is realised effectively on the XC6200 architecture because of the use of dynamically reconfigurable LUTs in such ‘divide and conquer’ m -bit multiplication. The final on-chip layout of a processing element of the ilk described is shown in Figure 5. The reconfigurable LUTs are vertically interleaved (columns 0 to 6) to facilitate the addition of the partial results (a 12bit adder appears in columns. 7 and 8). Finally, the result of the multiplication is accumulated with the PE’s incoming y value by a 16-bit adder (cols. 9 and 10).

We now address how the proposed runtime system will support the inherently reconfigurable systolic array. The application levels submit template systolic array symbolic circuit. Such a design would most likely be preconfigured to some initial or default set of taps. Upon receiving this design, the system begins the process of making the circuit resident on the reconfigurable resource. The Virtual Hardware (VH) Manager attempts to ensure the co-existence with other resident circuits. At this point, the array may be transformed and is, in effect, being denied its default orientation, location or both. The Transformation Services would then provide any necessary modifications to the circuit topology as dictated to it by the VH Manager. At this stage, the full range of geometric and replacement/re-routing facilities of the Transformation manager may be called upon.



Figure 5 Layout of systolic PE

Under the instruction of the VH Manager (and after any initial transformation), the array may be made resident by the configuration manager, which rapidly converts the symbolic circuit representation to a programming stream. This stream is directed towards the low level device driver which interacts directly with the hardware to physically configure the FPGA.

At this point, the application may submit a series of re-configuration requests, specifying the areas of the submitted circuit symbolically. With fine grain elements of reconfigurability, such as gate level in the multiplier LUTs, there is no need for transformation services. Additional write cycles may be used to update functionality. For coarser grain inherently reconfigurable designs, transformations may need to be applied. It is unlikely that an application supplied circuit would be immediately applicable in a circuit which has itself been transformed.

Given that the array is now resident, the remaining two main events involve the supply and recovery of data. Down-flow of data to the array may be managed as a style of reconfiguration. Application supplied data is likely to be retained in registers of the submitted design. Registers have state i.e. *configuration*. Therefore, just as topology reconfigurations are submitted, so are register *state* reconfigurations. Utilisation of features such as map-registers may be possible if symbolic representations are rich enough to allow the runtime system to identify such features. Applications may refer symbolically to register “bar”, and supply a configuration for it. The symbolic register identification is interpreted in the VH Manager which then directs the configuration manager to effect the desired state change.

Upflow of information, likewise, may be done symbolically. Just as an application is aware of the symbolic representation of its transformed design, it may also enquire as to the configuration state of a symbolically named register. The VH Manager once again interprets the symbolic identifier and directs the configuration manager to acquire the current configuration of the appropriate register components. The VH Manager, upon receipt of configuration state, replies to the application levels appropriately.

We recognise that circuits may not always be passive entities and may themselves actively seek communication with the application levels. To facilitate this, we propose an exception mechanism which allows the on-chip circuits raise, possibly parameterised, exceptions in the application levels. The VH Manager itself may be aware of circuit exceptions through symbolically defined exception register regions in the circuit.

2.4 Summary

The objective of our dynamic reconfiguration run-time system is to provide a high level interface to applications that wish to perform complex reconfiguration and circuit manipulation tasks. We view our run-time system as an operating system for the FPGA which executes on the host computer. From the case studies above, we derive the following general requirements:

- The ability to *reconfigure* an FPGA and access board-level resources (e.g. clocks) without directly communicating with a device-driver.
- The ability to *reserve* a chunk of FPGA resource. This allows one FPGA to be shared amongst several tasks and applications.
- The ability to *transform* a circuit e.g. change it's orientation or translate its position.
- A rich high level symbolic representation for circuits retaining the ability to quickly effect changes in the representation and transform the representation into device specific programming data.
- An architecturally correct representation of the FPGA device to support core algorithmic operations within various entities of the proposed system.

These requirements have influenced the design, in the following sections, of more detailed aspects of the proposed system.

3 System Overview

Figure 6 illustrates the data flow through the RAGE run-time reconfiguration system. The **virtual hardware manager** coordinates the execution of the other system components. This component is further described in section 4. The **device driver** hides the programming interface of the FPGA and PCI board (Figure 1) and presents a low level foundation on which more complex functionality is built. It virtualises the board's I/O ports and FPGA's configuration memory, enforcing

mutual exclusion and also maps the configuration memory into the host's address space. The device driver is described in section 7. The **configuration manager**, described in section 6, links the virtual hardware manager and the device driver. A programming *stream* is generated and this is mapped transparently to the programming memory of the FPGA.

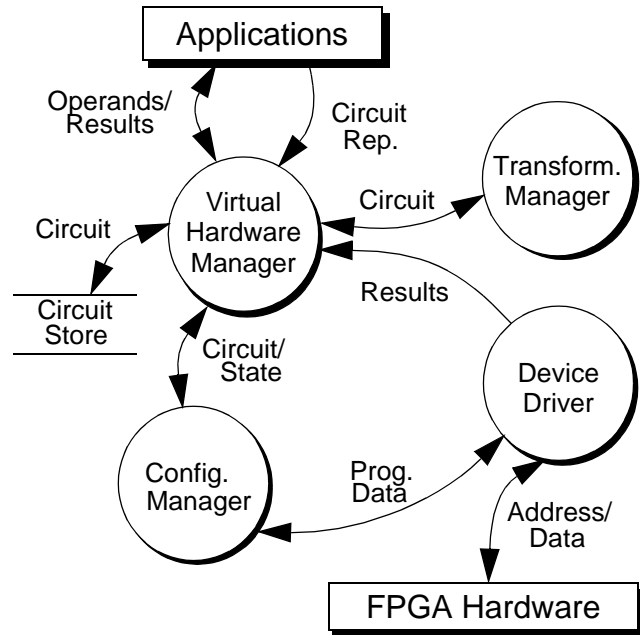


Figure 6 The RAGE System Dataflow.

The proposed system is designed initially to specifically operate with Xilinx XC6200 FPGAs [14]. This limits the direct applicability of our approach to one particular series of FPGAs. Despite this, the system has been designed to ensure many of the services make sense in other SRAM based FPGAs. We believe that one should gain experience in designing a good system for one device, and then try and generalise for other FPGAs. We expect much of the system to be applicable to the Atmel 6000 series/National Semiconductor CLAY FPGAs.

The XC6200 series of FPGAs are based on a grid of cells (48x48, 64x64, 96x96 or 128x128) each of which can realise routing, any one or two input logic function or a 2-to-1 multiplexer. These functions can be combined with a register. Cells may communicate with their neighbours, or to cells further away using a hierarchical routing system based on blocks of size 4, 16 and 64. It is possible to reprogram individual bits of configuration memory, or to program 4 cells in one 32-bit write cycle (more cells can be reconfigured at once if there is any regularity in the design). The programming interface runs at 20MHz. A key feature of the XC6200 architecture, which is exploited by our system, is the ability to read and write directly to cells configured as registers on the FPGA. This

means that inputs do not need to be explicitly routed to circuits from I/O pads, and provides a very powerful location transparency which allows us to realise relocatable circuits.

4 Virtual Hardware Manager

The virtual hardware (VH) manager provides the main interface between the application and the run-time system. Calls are made to the VH manager to make a circuit resident, communicate with that circuit, and free its resources afterwards. A typical sequence of events might be:

- Application submits circuit in an external representation to the VH manager which converts this into an internal circuit data structure which is put into the circuit store.
- Application requests circuit to be made resident.
- VH manager checks current resource utilisation and finds a location for circuit, possibly after having to pass it to the transformation manager for translation.
- VH manager passes the, possibly transformed, circuit to the configuration manager for conversion to programming data and download to FPGA via device driver.
- Application provides circuit input data to VH manager and requests execution.
- VH manager downloads the data to the circuit, sets up interrupt handlers for completion and signals the circuit to begin execution (all through the configuration manager/device driver).
- On completion interrupt, the VH manager reads back the result data and passes it up to the application.
- Application signals that the circuit is no longer required.
- VH manager marks circuit resources as free and updates the configuration manager.

During this sequence of events, the application might make other requests to partially reconfigure the circuit or download other circuits. Such requests are done at a high-level using symbolic components and circuits. The VH manager maintains a map of the FPGA usage in terms of these high-level circuit blocks, relying upon the configuration manager to map these to actual FPGA cells. For example, the VH manager might refer to a register of a circuit as `counter.preload` whereas the configuration manager would understand this as being cells 4-20 of column 12 on the FPGA.

5 Transformation Manager

We cannot rely on a circuit always being suitable for residence on the reconfigurable device. Allowing the reconfigurable device to be shared amongst circuits means that the ‘default position’ of one circuit may already be in use by another. To facilitate a flexible virtual hardware system, circuits themselves must be flexible elements. If a circuit cannot be placed in its desired default position, then the virtual hardware system must be able to somehow transform the circuit

into a state in which it can be made resident in some other section of the reconfigurable device. The entity which is responsible for providing these facilities is the Transformation Manager.

5.1 Transformation Services

RAGE transformation services use a subset of two dimensional graphical transformations. These include translation, rotation, mirroring and scaling. Used as a set of primitive operations, it is possible to construct complex compound transformations which allow an entity to be manoeuvred around a two dimensional space. Furthermore we identify the usefulness of composition transformations such as “join” and “slice”.

Our system holds a conceptual model of a 2D reconfigurable area and hence the standard 2D geometric transformations are directly relevant.

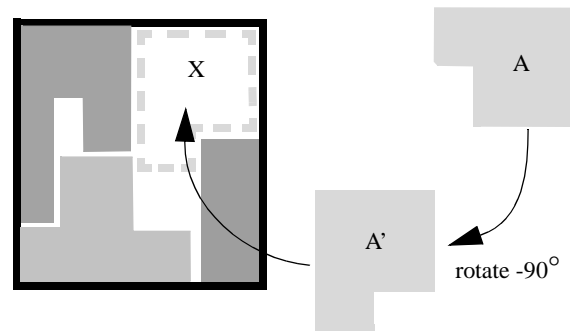


Figure 7 Transformations to support efficient device usage

Consider, for example, the use of geometric transformations to place the circuit A, in Figure 7. It is clear that there is in fact no space readily available for the circuit to occupy. By imposing a translation and rotation transformation, it is possible to have the circuit resident in the area of free space marked X. Without such functionality, the circuit could not have been made resident on the FPGA in its current state. The alternatives would have been either to deny service to the application requesting that the circuit be placed on the FPGA, or to allow the runtime system to remove resident circuits from the FPGA.

Evicting a circuit is a viable alternative to transformations and may be less expensive in terms of raw costs. Since circuits rely on uninterrupted slots of residency to reach cost/benefit break even points, transformations are however more attractive and less disruptive. By composing transformations the cost may be reduced to a level lower than that of pre-emption.

5.2 Transform Support Issues

We have identified a use and justified a need for transformation services within the context of our runtime system supporting virtual hardware. In this section we consider some of

the issues involved with supporting a set of 2D geometric transformations within the context of our system.

At some later point, within the configuration manager, this transformed symbolic circuit representation is translated into FPGA specific programming information. The effect of transformations, therefore, is to modify the contents of a symbolic data structure representing a routable circuit. A netlist style representation which is also capable of explicitly defining routing information and indicating placement positions is used for this purpose.

We recognise that transformations may require modifications to the routing and/or placement decisions made at circuit compile time. The resources available on the Xilinx XC6200 are generally regular, but possess some irregular, asymmetric, features. Compile-time routing makes decisions based on a particular placement of the current circuit and assurances of the availability of resources and the general “lay of the land” upon which the circuit will be made resident. Transformations at runtime are likely to invalidate some of these assurances. In such situations, the circuit will most likely become invalid and, if made resident on the board, fail to perform its intended function. Indeed, the circuit may possibly interfere with the operation of other resident circuits as the routing bounding box is known to exceed the placement bounding box of a circuit in some instances (this is an issue for the virtual hardware manager, however, which has to decide what constitutes the footprint of a circuit.).

We believe that re-routing parts of a circuit at runtime is likely to be necessary on the Xilinx XC6200. Consider a circuit of dimensions 3x4 located at position (0,0) on the device. If this circuit were to be translated to position (1,1) on the board, then its internal routing would have to be routed across a series of switching matrices that exist on the boundaries of every group of 4x4 cells. These switching matrices facilitate access to the hierarchical routing resources that are particular and novel to the XC6200. It is clear that the internal routing details of the circuit now have to be adjusted.

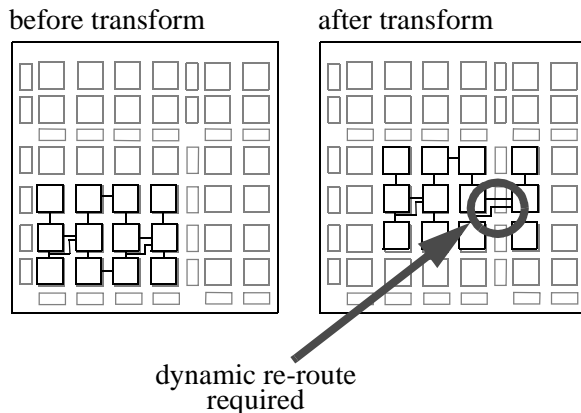


Figure 8 Circuit Invalidation by Transformations

Irregularities in the resources in the target FPGA also introduce other scenarios which must somehow be dealt with. The XC6200 architecture possesses a series of “hidden” inverting multiplexors. These features constitute an irregularity in the resource and a circuit designed, placed and routed at position (10,20) may not be placed at (21,10) as the irregular features present at (10,20) are not present at (21,10). The resolution of such resource clashes, for the XC6200 architecture, involve introducing new components into the circuit. Introducing new components into a circuit, however, may have a knock on effect on the placement and the availability of routing resources for other components.

To resolve these problems, we identify three possibilities, all of which rely on the ability of the system to detect when a circuit transformation will invalidate the circuit. The first resolution is stringent and involves refusing the current set of transformation requests. At first glance this system may appear pointless, and all flexibility may appear to have been lost. It is possible, however, for the system to operate within this framework if the use of transforms is restricted to adhere to the symmetry and regularity of those available resources. If this convention is followed, then the circuit shown in Figure 8, above, would never be translated to the position in which its internal routing would have needed to be altered. Using this method, we can extend the applicability of a pre-placed and pre-routed circuit, but at the cost of granularity of placement and therefore, overall, at some cost to the flexibility of the system as a whole. Similar restrictions can be applied to the other transformations.

Alternatively, and at the opposite extreme, the system may actively attempt to re-route and/or re-place components as required to overcome the resource clash. Runtime full place and route is likely to be unfeasibly slow. Rather than attempting to support a full place and route engine, we suggest a bounded runtime place and route which attempts to localise changes to placement and routing. Through a process of localisation and problem minimisation the duration of runtime place and route on a symbolic circuit representation may be greatly reduced. Additional mechanisms to reduce the duration of runtime fitting onto the reconfigurable resource would be to abandon the search for an optimal solution to the placement or routing of a particular circuit. By taking only those transformational steps which make the circuit valid for residence and no more, it is possible to simplify runtime place and route. Furthermore, it may be possible to gradually evolve a better routed/placed circuit during the course of the application's lifespan.

In theoretical terms, just as a program in memory exhibits locality of execution and locality of reference, some circuits in a system will show locality of placement and therefore locality of routing. Given the possibility of this behaviour, evolutionary runtime placement and routing would preserve maximum flexibility of the overall system

and work towards reducing the transformation service overheads to a relatively acceptable minimum.

6 Configuration Manager

Device drivers need to be lean and use as few resources as possible. They provide a high degree of device independence but software layers built on top, usually code libraries, provide the final device abstraction. Similarly, the configuration manager of the RAGE system acts as the final abstraction layer, providing a device independent interface to the VH manager. Here is a partial list of the services to be provided by the configuration manager:

- Configure a circuit;
- Partially reconfigure an existing circuit;
- Load data, passed from the application, though the VH manager, to registers in an circuit resident on the FPGA;
- Pass state information, provided by the device driver, to the VH manager, e.g. an interrupt message if the FPGA device overheats;

The symbolic representation of circuits must be converted to the programming stream for the FPGA concerned. Likewise, the data to be loaded into the circuit registers must be in the correct format. To this effect, the configuration manager must preserve a mapping between the two representations. This allows a request from the VH manager to be converted into the correct programming data for the FPGA. Similarly, state data from the device driver, such as interrupts, may be passed back to the VH manager using a message indicating that the counter circuit had run for the requested number clock ticks.

Rather than waste valuable programming cycles querying the FPGA to obtain configuration and state data, the configuration manager maintains an image of the FPGA configuration and state. It provides a query service to the virtual Hardware Manager.

7 Device Driver

The device driver provides a set of functions which enable the configuration manager to program the FPGA device and communicate with a PCI board containing one or more FPGAs. The required functionality includes:

- Writing and reading the PCI board SRAM;
- Writing and reading FPGA cell programming data;
- Interrupting the host when certain board generated events occur;
- Setting the FPGA clocking frequency;
- Monitoring the current drawn by the FPGA;
- Clocking FPGA circuits with individual, continuous, or a preset number of clock cycles.

Both the FPGA programming interface and the SRAM are transparently mapped into the host processor's memory space. Programming the FPGA and writing to SRAM are simple programming language assignment operations.

The SRAM programming interface provides a fast cache which can be used by both the host and the FPGA. There are two banks of SRAM and although a bank cannot be shared between host and FPGA, ownership of a bank can be switched by the device driver to permit data sharing.

A total of three interrupts can be generated by the PCI board. The first interrupt is triggered when the current drawn by the FPGA exceeds a predefined threshold. The second is triggered when the last in a preset sequence of clock cycles has elapsed. The third interrupt can be requested by the FPGA for any reason. The device driver is interrupted, and may either handle the interrupt or pass relevant information to higher layers of software allowing the application to take appropriate action.

8 Application-System Level Interface

The system entities we described in earlier sections do not cover all possible application level requirements. Rather than being inflexible, however, our proposed runtime system is intended to provide a general foundation, applicable to many general application contexts. Our system can be extended by inserting application specific entities as mediators between our proposed basic system and the application.

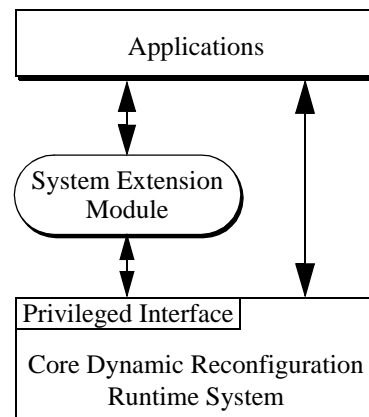


Figure 9 Core System Extension by Application-Context Specific Module.

An example of such system functionality extension would be partial evaluation, as discussed in section 2.2. An alternative and simpler example is a Specialisation Module effecting constant propagation—a standard optimisation performed at compile time [13]. Known inputs are fed through the circuit and logic optimisation is performed e.g. an AND gate with one high input can be replaced by a wire driven by the second input. Although this not core system functionality, it may be profitable, in some application contexts, to perform this kind of optimisation at run-time for dynamically reconfigurable FPGAs. This is especially true if the specialised circuit will be used many times, thus recouping the run-time

cost of calculating it. For example, consider the AND cell in Figure 10(i) with a constant high input which, after some calculation, can be optimised to Figure 10(ii).

A Specialisation Module would take a circuit description and a set of protected input-register values and propagate the values of these ‘constant’ registers throughout the circuit. Propagation stops when an output register is encountered, or a flip-flop or asynchronous feedback path is found.

An important consideration, at this point, is the means by which an Extension Module communicates with the core system. The core system presents a simplified interface to applications. A privileged interface, which is a superset of the application interface, would allow Extension Modules mediated and controlled access to internal runtime system data structures. By this means, an Extension Module may acquire detailed information about the transformed state of a previously submitted circuit as well as having fast access to data structures.

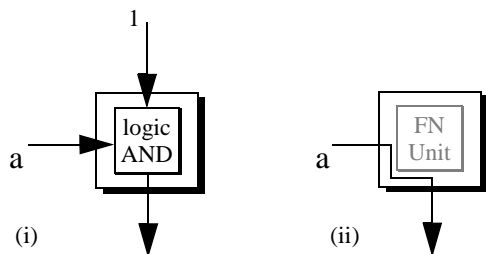


Figure 10 Simple constant propagation of logical “And”

9 Related Systems

At present there are very few FPGA run-time reconfiguration systems. All available systems are highly specialised to particular applications or groups of applications. RAGE is a more generic system and is tied to no particular set of applications. Applications using RAGE must be designed with the system in mind but are not tied to specialised development tools. The RAGE system can be used by simply linking its exported functionality to a library of functions. RAGE is not a development system. It allows access to the FPGA hardware using a high level interface. The goal is that the design is sufficiently flexible to support other applications outside of the ones being examined by the RAGE group. In the following sub-sections, we discuss how our proposed system applies to other applications in the field.

9.1 The Dynamic Instruction Set Computer (DISC)

The Dynamic Instruction Set Computer (DISC) developed at Brigham Young University [12] exploits FPGA reconfiguration to dynamically supplement its fixed instruction set. Each instruction or circuit module can be swapped on and off the CLAY31 FPGA as the running application demands. Like the RAGE system, DISC maintains a library of circuits and DISC circuits are designed with a global context (the system’s view

of the FPGA) which allows them to operate at any vertical position on the FPGA. Circuits designed for RAGE, however, can be placed anywhere on the FPGA. Dynamic reconfiguration on DISC is achieved by a reconfiguration controller which executes on the FPGA. In contrast, the RAGE system runs entirely on the host machine which allowing easier debugging. Software/hardware co-design with DISC requires the use of **lcc**, a retargetable C compiler. The RAGE system provides a higher level interface for hardware/software co-design. Programs using virtual hardware need only call the system functions, which are linked in to the application program.

Despite these differences, most applications running on DISC could be easily adapted to RAGE by providing a group of circuits placed and routed for the Xilinx FPGA and by including library function calls within application code.

9.2 The Run-Time Reconfigurable Neural Network (RRANN)

RRANN [2] uses dynamic FPGA reconfiguration to implement the three stages of the neural network backpropagation algorithm. Each stage of the algorithm is represented by a circuit module which is swapped on and off the FPGA hardware as demanded by the application. Only one circuit module is resident on the hardware at a given time. Some static circuitry is resident; this controls dataflow and the sequencing of execution of the dynamic modules.

RRANN2 [3] takes this simple swapping of circuits one stage further by using swapping at finer granularity. By maximising the static circuitry, less time is spent reconfiguring circuits, and system efficiency is increased. Smaller circuit elements are reconfigured, for example a counter is shaved from 11 to 8 bits, and the rest of the circuit remains unchanged.

Either of these applications can be easily adapted to use RAGE. By creating a library of circuits known to operate on the Xilinx FPGA and by writing code in any language from which Windows DLL functions can be called, the application can have access to the virtual hardware without resorting to recompiling the whole program.

9.3 Fast Reconfigurable Crossbar Switching in FPGAs

This technology, developed at the Department of Electrical and Electronic Engineering in the University of Strathclyde [1] implements a fast reconfigurable crossbar switch on an Atmel AT6005 FPGA. The switch is part of an ultrasonic imaging system which is highly time dependent. The circuit design is tailored specifically for the Atmel FPGA.

This application’s dependence on the Atmel FPGA may preclude its implementation on the Xilinx FPGAs. RAGE, however, will eventually support a variety of FPGA architectures and so it is likely that this application could be adapted to run on RAGE. The dynamic circuit reconfiguration would

be handled by the virtual hardware manager using pre-placed and routed alternative circuits from the circuit store or may employ the services of the RAGE transformation manager.

10 Summary

As more applications take advantage of dynamically reconfigurable FPGA-based hardware, there is a need for software support to facilitate dynamic reconfiguration and to provide high level abstractions of hardware which allow it to be shared amongst many tasks. We have described the architecture of a proposed run-time system that meets these objectives whose design was motivated by our own experiences and applications of dynamic reconfiguration.

We wish to acknowledge the support of Xilinx Corp. for software, hardware and technical support.

This research is supported in part by UK EPSRC grant number GR/K82055.

References

- [1] H. Eggers, P. Lysaght, H. Dick and G. McGregor, *Fast Reconfigurable Crossbar Switching in FPGAs*. In, R. W. Hartenstein, M. Glesner (Eds.) *Field-Programmable Logic—Smart Applications, New Paradigms and Compilers*, Springer Verlag, Germany, 1996, pp. 297-306.
- [2] James G. Eldridge, Brad L. Hutchings. *RRANN: The Run-Time Reconfiguration Artificial Neural Network*. IEEE Custom Integrated Circuits Conference. 1994.
- [3] J. D. Hadley, B. L. Hutchings. *Design Methodologies for Partially Reconfigured Systems*. FCCM'95. IEEE Computer Society, 1995.
- [4] J. Hogg. *A Dynamic Hardware Generation Mechanism*. In, M. Sheeran, S. Singh (Eds) *Designing Correct Circuits*, Electronic Workshops in Computing, Springer Verlag, 1996.
- [5] T. Kean, B. New, B. Slous. *A Multiplier for the XC6200*. Sixth International Workshop on Field Programmable Logic and Applications. Darmstadt, 1996.
- [6] H. T. Kung. *Why Systolic Architectures*. IEEE Computer. January 1982.
- [7] Charles E. Leiserson. *Systolic and Semisystolic Design*. IEEE Conference on Computer Design/VLSI In Computers (ICCD'83). 1983.
- [8] Satnam Singh and Pierre Bellec. *Virtual Hardware for Graphics Applications using FPGAs*. FCCM'94. IEEE Computer Society, 1994.
- [9] Satnam Singh. *Architectural Descriptions for FPGA Circuits*. FCCM'95. IEEE Computer Society. 1995.
- [10] Satnam Singh, Jonathan Hogg and Derek McAuley. *Expressing Dynamic Reconfiguration by Partial Evaluation*. FCCM'96. IEEE Computer Society. 1996.
- [11] M. Sheeran, G. Jones. *Circuit Design in Ruby*. Formal Methods for VLSI Design, J. Stanstrup, North Holland, 1992.
- [12] Michael J. Wirthlin and Brad L. Hutchings. *A dynamic instruction set computer*. FCCM'95. IEEE Computer Society. 1995.
- [13] Michael J. Wirthlin, Brad L. Hutchings. *Improving functional density through run-time constant propagation*. To be published in FPGA'97.
- [14] Xilinx. *XC6200 FPGA Family Data Sheet*. Xilinx Inc. 1995.