Resource-Aware Run-time Elaboration of Behavioural FPGA Specifications

U. Malik, K. So, O. Diessel
Computer Science and Engineering
University of New South Wales
{*umalik,keiths,odiessel*}*@cse.unsw.edu.au*

Abstract

The Circal process algebra is being used to explore the behavioural specification of systems that are mapped to field programmable logic circuits. In this paper we report on the implementation and performance of an interpreter for system specifications given in the Circal language. In contrast to the typical design flow for field programmable technology in which designs are statically partitioned, synthesised, and mapped to pre-allocated resources, in this system the specified circuits are extracted from behavioural specifications that are partitioned, elaborated, mapped, and configured at run time as control passes through them. We report on the details of a design that targets the Celoxica RC1000 co-processor and assess preliminary performance results for this implementation. The results clearly demonstrate our method is a practical approach to overcome resource constraints, particularly in applications where these change at run time. The results also establish a benchmark against which to measure future improvements and alternative methods.

1. Introduction and related work

The Circal process algebra is a formal language suited to the high-level specification, analysis, and construction of hierarchical controller circuits. Historically, process algebras such as Circal [10], CCS [12] and CSP [7] were used for analysing and verifying properties of concurrent systems. These features were then employed in the design and verification of VLSI hardware, and led to the investigation of the use of the CSP-derived language Occam as a specification language for FPGAs [14]. These early steps occurred at a time when computing using programmable logic was beginning to grip a section of the research community and thus competed with VHDL and a raft of alternative methodologies that were proposed to solve the problem of readily specifying and automatically generating efficient circuit designs for programmable logic devices and systems.

At present there are several FPGA design paradigms including JHDL [1], Handel-C [13], PAM-Blox [8], and Lava [2], that principally target the design of mainstream data-intensive FPGA computations such as signal and image processing, encryption, and streams processing. These paradigms generally expand on conventional imperative approaches towards describing hardware structures, and adhere to a relatively static view of the target architecture. Reconfiguration is catered for via the off-line production of bitstreams for static partitions that can at best be swapped during the execution period of an application using purpose-designed, application-specific controllers. In order to produce efficient designs, developers need to be familiar with both digital design techniques and target technologies, and the dynamic nature of current programmable technology is not readily exploited. These factors have motivated us to explore an alternative design approach with the goal of exploiting the potential of dynamically reconfigurable hardware and providing high-level semantics for describing both static and dynamic functionality that can automatically be mapped to field programmable devices [6, 11].

Prior work in the use of Circal as a behavioural specification language for FPGA circuits resulted in the development of a compiler that produced static circuit designs targeting the Xilinx XC6200 device family [6]. In order to cope with mapping large circuits to limited FPGA area, methods for virtualizing the compiled designs were investigated and resulted in the design of the interpreter reported for the XC6200 circuit model in [5]. In tandem, the compiler was re-targeted to the Xilinx Virtex family and implemented on an Annapolis Microsystem's Wildcard [15]. This paper reports on the implementation of the interpreter design targeting the Virtex XCV1000 chip embedded in a Celoxica RC1000 card.

The ability to reconfigure circuits at run-time, as offered by contemporary field-programmable devices, is not used by our interpreter to increase the performance of applications over static implementations, but rather to overcome resource limitations. We believe that our experimentation with hardware virtualization will give us a better understanding of the costs involved in managing large partitioned

computations at run-time and hence allow us to gauge the effectiveness of using programmable-logic as an alternative implementation technology.

We present an assessment of the performance of our method and draw some conclusions about the efficacy of our techniques. Work is going on in generalising the specification language and the hardware mapping so as to implement the full specification of the Circal process algebra. Not only do we then expect to be able to explore in-circuit verification of designs, we also anticipate expanding the descriptive scope of Circal to encompass dynamic structures and behaviours.

We conclude this section by describing two scenarios where we see something akin to the Circal behavioural specification paradigm in general, and automated hardware virtualization in particular, being of use in the design of FPGA circuits.

Ever increasing device densities open up the possibility of managing FPGA tasks onto the chip itself. In such a scenario it is expected that the chip area would be divided into partitions consisting of user and operating system space respectively. As most of the operating system tasks are control oriented there is increasing need for mapping general controller circuits (i.e. ones that can be modelled with composed finite state machines) to reconfigurable logic. Moreover, since the operating system will only have access to limited chip area (as we typically want as much user space as possible) we envisage virtualizing the operating system area itself. The techniques presented in this paper could be employed to realise this goal.

In order to implement recursive computations in hardware, mechanisms for dynamically allocating and recovering used hardware resources are needed. These functions require sophisticated automatic control, are ideally specified in an abstract, high-level fashion, and demand some form of run-time design and implementation of circuits. Not only is Circal suited to the behavioural specification of such recursive dynamic processes, the Circal interpreter marks a beginning in the development of automated facilities to support universal computations in programmable hardware.

Section 2 provides a background to Circal and introduces the concept of a Circal interpreter. Section 3 details the design, implementation and operation of the interpreter. Section 4 presents an assessment of the interpreter's performance, and Section 5 presents our conclusions and outlines directions for future work.

2. A Circal interpreter

2.1. Circal as an FPGA specification language

State transition diagrams, automata and similar models are widely used to model simple dynamic behaviours of systems. Circal extends these concepts by introducing structural and behavioural operators. Structural operators allow the decomposition of a system in a hierarchical and modular fashion down to a desired level of specification. Behavioural operators allow us to model the finite state behaviour of the system where state changes are conditioned on occurrences of actions drawn from a set of events. This set is called the sort of the process. Refer to [9] for a detailed description of the Circal process algebra.

Circal processes can be looked upon as interacting finite state machines where events occur and processes change their states according to their definitions. These processes can be composed to form larger systems with constraints on the synchronisation of event occurrence and process evolution. Given a set of events, all composed processes must be in a state to accept this set before any one of them can evolve. If all agree on accepting this set, they all simultaneously evolve to the prescribed next state. This constraint can be relaxed through the use of an abstraction operator (encapsulating the event) or relabelling the event to a different name. Events that are not in a process's sort are ignored.

In the static compiler for Circal, the circuits are produced as blocks of modules that reflect the hierarchy and interconnection of the system. The blocks implement individual processes interconnected via global synchronisation logic. The individual process blocks themselves are decomposed into blocks that reflect the system hierarchy. Within a process block the finite state machine behaviour is implemented using a finer grained rectangular block structure where each component implements some single logic function. Refer to [6] for the details of the mapping of Circal operators to digital circuits.

2.2. Interpreting Circal specifications

Circal can be used as a language for programming hardware computational structures. For a system realising Circal specifications on hardware to be generally useful it should support large and dynamic behaviours. In this respect, the current compiler is restricted by the size of the available hardware and can only handle static processes.

Traditionally, the hardware size constraint is handled by translating the specification into a netlist, then partitioning the netlist either manually or semi-automatically, before finally producing configuration bitstreams for each partition. At run time, the system selects the appropriate bitstream and loads it onto the chip. This technique is limited in the sense that it cannot deal with dynamic structures, i.e. processes with time-varying hardware needs. Moreover, it cannot exploit changing hardware availability to harness parallel computation. In order to address these issues we have developed an interpreter that can respond to run time conditions.

An interpreter is a system that realises an algorithm on an architecture, as directed by the flow of computation. This allows adaptability to dynamic computational structures. Traditionally, interpreters are used as virtual machines to execute the same code on different hardware platforms. However, an interpreter can also provide a virtual hardware environment by exploiting the temporal locality of computation. In other words, as opposed to a compiler that translates a computation completely before execution, an interpreter only needs to implement the required part of the computation while storing the rest in a suitable format. At run time, the required portion of the computation can be implemented on the target architecture. In this manner much bigger computations can be realised.

The Circal interpreter enhances the existing compiler by incorporating virtual hardware management facilities. Whereas the Circal compiler derives a monolithic circuit and loads it onto an FPGA in a single configuration, the interpreter elaborates and loads parts of the circuits as they are needed. The issues to be addressed are:

1. Determining the granularity of a partition that can be executed. If we look at microprocessors as interpreting an instruction set architecture (ISA), resources to implement at least one instruction are always available.

2. Developing the techniques to cope with the situation where the required part of the computation does not fit on the target architecture. The microprocessor solution is to time multiplex hardware.

In Circal, even though all processes are updated simultaneously, the transitions within a process are strictly sequential. This means, at any time, a process only needs enough hardware to implement a single state's transition logic. Hence, the unit of computation is a single process transition. The actual size of the hardware is, of course, determined by the requirements of the largest single state in terms of the number of possible transitions. Currently, we assume that the logic for at least a single state of all processes can be implemented. However, the Circal interpreter implements as much logic of each process as is permitted by the available area. This results in:

1. Reduced reconfiguration costs if we overlap reconfiguration with execution.

2. The possibility of adapting to time-varying hardware availability.

In the current implementation, hardware size is kept fixed, with reconfiguration and execution as disjoint phases. Currently, the interpreter only implements static processes. However, we hope that our techniques will guide us towards methods for handling dynamic processes as well. We now describe the Circal interpreter in detail.

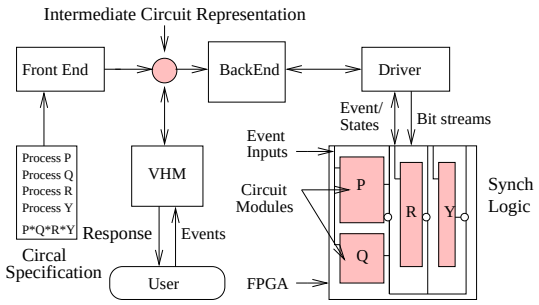


Figure 1. An overview of the interpreter

3. Interpreter operation

The interpreter improves on the compiler in the following ways:

1. Whereas the compiler parses a Circal specification and directly produces the bitstreams, the interpreter first translates the specification into a state-transition graph representation.

2. The compiler treats the chip area as a single 2D block of cells within which process blocks are stacked in a 1D manner. This does not fully utilise the available chip resource. The interpreter, on the other hand, partitions the chip area into strips and allocates a pre-sized block to each process depending on its needs.

3. At runtime, the interpreter selects a subgraph of each process, where the size of the subgraph depends on the area allocated to that process. The selected subgraph is then interpreted to produce a bitstream similar to the existing compiler.

4. As processes evolve, different portions of their state-graphs are selected and physically realised. In this manner large specifications can be interpreted, thus automatically overcoming hardware limitations.

In order to achieve these goals, a virtual hardware manager (VHM) is inserted between the front and back ends of the static compiler to construct the Circal interpreter (see Figure 1). After initialisation, the VHM interacts between the user and the back end, while implementing the above-mentioned functions (see Figure 2). These functions are described below in more detail. For a complete description of these functions, refer to [5].

3.1. Intermediate circuit representation

The specified circuits are modelled as state-transition graphs. This representation is aligned with the interpreter's

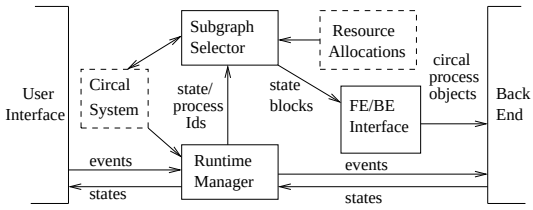


Figure 2. The VHM overview.

unit of computation and naturally follows from Circal's semantics.

In software, the data structure representing a single transition is called a state block. It consists of a state name and guarded pointers to other state-blocks, thus representing the transition graph.

The state block is the unit of partitioning. As many state blocks are selected as can be accommodated by the area allocated to that process.

3.2. Subgraph selection

The interpreter selects a subgraph of each process, at run time, depending on how much chip area is allocated to that process. This is done by traversing the graph rooted at the current state in a breadth-first manner. In the extreme case, only the current state of the process is implemented.

The function used to estimate the area depends upon the implementation of the subgraph components in the target architecture. For the Virtex implementation, given w configuration logic block (CLB) columns, the state subgraph must satisfy the following constraints:

$$\begin{aligned} \text{number of minterms} &\leq 2(w-1)-1 \\ \text{number of edges} &\leq 2(w-1)-1 \\ \text{number of states} &\leq \frac{w}{2} \end{aligned}$$

We define a boundary state to be a state that triggers a circuit update in the interpreter as its corresponding logic is not fully implemented.

3.3. Detecting the need for circuit swapping

During development, the system operates in a "debug" mode whereby a human user inputs event triggers and responds to the controller state, as opposed to the controller interacting with other systems in its embedded environment. Embedding the controller involves replacing the user with the controlled system which may or may not be on the same chip as the controller.

The interpreter gets events from the user and presents them to the system on chip. It then reads back the state and reports it to the user. Since the interpreter has only implemented a subgraph, there will be boundary states. A new subgraph must be implemented if the process enters one of these states.

Currently, we have implemented a polling mechanism that checks after each synchronisation pulse whether a boundary state has been entered. Before passing the next set of events to the implemented system, the interpreter determines, by looking at a table of implemented states, whether any of the processes has hit its boundary. If it has, it traverses its state transition graph as described in the last section. In an embedded system, the polling mechanism would be replaced by an interrupt mechanism that is triggered if a boundary state is entered by a process. This can be checked by associating a boundary status flag with each state flip-flop.

3.4. Chip partitioning

Chip partitions are static and are created during the analysis phase. The chip partitioner initially allocates sufficient space to each process to implement the behaviour for a single state, otherwise the system is terminated. Then it expands the initial partitions, if possible. The expansion factor is derived from an analysis of the possible expansion of the process blocks if the interpreter implements more states. Circal processes are therefore each allocated their own region of the chip. These are then wired together using a coarse-grained parallel wiring harness [3].

3.5. FPGA circuit design

The interpreter targets the Celoxica RC1000 co-processor board [4], supplied with a Xilinx Virtex XCV1000 FPGA [16].

Each process is allocated a rectangular block of CLBs to implement an active subgraph on the FPGA. These circuit blocks are collated into vertical stripes on the FPGA as depicted in Figure 3. The layout attempts to amortise the reconfiguration cost when more than one process needs to perform a circuit update. Since a configuration frame defines behaviour over an entire column in the Virtex, the configuration cost in terms of frames to be reconfigured will be reduced if some of the processes being updated belong to the same column.

In our current implementation, only a single composition of processes is supported. The synchronisation logic consists of an AND gate for each column of processes; and a root AND gate forms the conjunction of these column synchronisation outputs that is then distributed to every process on the FPGA.

The layout of intra-process circuits follows a similar design rationale. The allocated area is sub-divided into smaller rectangular blocks to implement different functional circuits as shown in Figure 4. The Minterm block determines whether a set of events that triggers a transition is supplied. The Guard block then computes the possible transitions that are triggered. The Requester block checks whether the process is in one of the source states accepting the set of supplied events. If so, the Asserter block asserts the synchronisation request signal and the destination state of the transition. The States block stores the sub-process states in a one-hot encoding and has combinational logic to accept a synchronisation acknowledgement signal. Refer to [15] for a detailed description of the functionality and layout of a process logic block as implemented in the Virtex compiler.

Most of the combinational logic is laid out in single-slice modules for two reasons: it exploits the dedicated fast-carry chains present in the Virtex architecture and it minimises the number of configuration frames required for a reconfiguration. The state block is allocated a single CLB column, and all state blocks of processes in the same column are aligned so that the state of an entire column can be read with four partial readback frames.

The layout uses complete routing between all logic blocks thereby implementing a fine-grained version of the parallel harness structure proposed in [3]. These routes are static and do not change during subsequent updates. The advantage of using static routing is that it avoids the dynamic un-routing and re-routing during circuit updates, which are slow and contribute a significant overhead to the reconfiguration time. All routes between the logic blocks are fully routed during the generation of the initial subgraph so that only changes made to the bitstream during circuit updates are changes to the LUT contents.

Since the layout is deterministic, the interpreter can determine the subgraph that will fit in the allocated area. JBits [17] was used as the interface to program and reconfigure the FPGA. During a circuit update, the interpreter software reconfigures as many LUTs as necessary using object references saved during circuit initialisation.

4. Timing analysis

For a system that dynamically elaborates FPGA designs, run-time performance is of main concern. In the case of our interpreter, the time it takes to select a subgraph at run-time and to implement it onto the chip are crucial factors in determining its performance. Since these two are distinct phases, we can measure their timings separately. We first report the subgraph selection performance and then its implementation on Virtex.

The subgraph selector (SGS) is the main component of

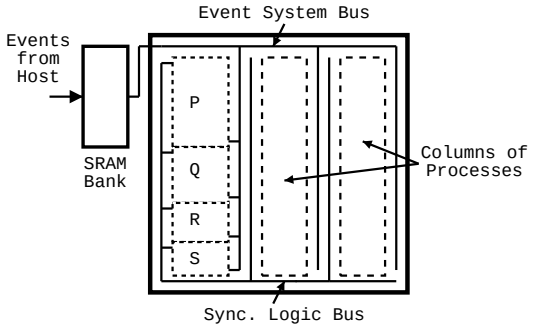


Figure 3. The partition of the FPGA into process blocks

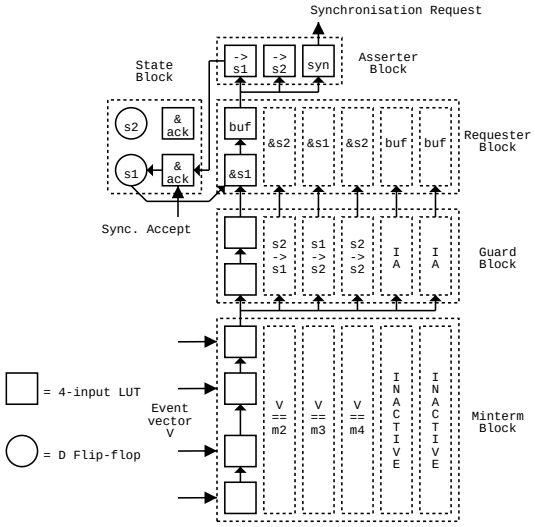


Figure 4. The internal layout of a process block

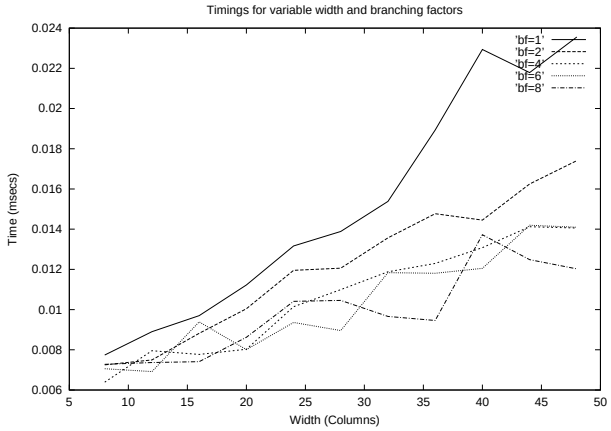


Figure 5. Run-time subgraph selection

the VHM. Given a state, the time it takes to reach the next states that will fit into the available area determines the execution time. Hence two factors influence the run time of the SGS; the branching factor of the specification graph and the size of the partition available to that process.

As the branching factor increases, more next states are reachable in one step if the data structure implementing the graph permits this operation. The interpreter takes a constant amount of time to extract the next level, as all pointers to the next states are stored in the state block. Hence for a fixed partition size, the higher the branching factor, the quicker the SGS will select the states that will fit and vice versa.

The partition size determines how many states can be implemented in that area. As the space available to a process is increased, more states are expected to be selected and hence a higher running time should be observed.

These predictions were confirmed by implementing a large number of randomly generated single process specifications with fixed branching factors and averaging 500,000 graph selections to obtain the plots illustrated in Figure 5. Here, only CLB column width is considered as the SGS uses this parameter for determining the amount of the graph to select. The system used for extracting the data was a Linux-based Java system running on a 1.8 GHz Pentium-IV.

The technology dependent backend was tested on a 1.6GHz Pentium-IV system running Java under Windows 2000. The initialisation time refers to the time taken to generate the bitstream from the initial Circal subgraph. The circuit update specification time refers to the time take to generate an updated bitstream from a new subgraph of the same process. The partial reconfiguration run-time is the time needed to load or partially reconfigure the FPGA. The relationship between FPGA circuit width and these run-times are shown in Figure 6.

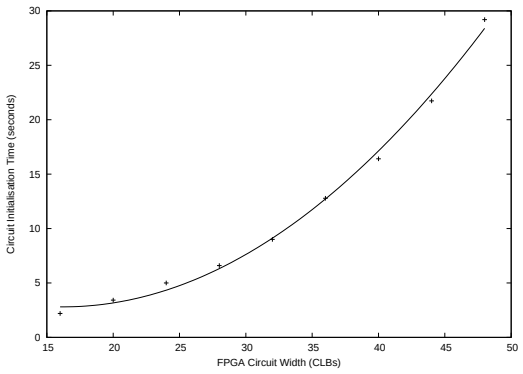
These results show a $\Theta(n^2)$ relationship between circuit width and run time. For the initial bitstream generation, this occurs because the parallel harness wiring structure, consisting of fanout routes across the width of the circuit, needs to be routed for a number of circuit modules proportional to the width of the process block area. For the update bitstream generation, a number of LUTs proportional to the area of the FPGA circuit has to be changed. In the layout schema, the height of the circuit area is proportional to the width, causing the correspondence recorded. The reason behind a similar relationship between partial reconfiguration time and circuit width needs to be further investigated, as it is expected that the partial reconfiguration time should hold a linear relationship with circuit width due to our layout schema and the column based nature of the Virtex configuration frames.

The initial bitstream generation is significantly longer than the update bitstream generation due to the router runtime at initialisation. Indeed, a complete configuration takes at least 31ms on our system. Circuit update times are in subsecond domain for the circuit sizes tested. It may be possible that a bottleneck of programming configuration bitstreams lies in performing bit-oriented manipulations of the configuration bitstream in JBits that operates under a Java virtual machine model of computation.

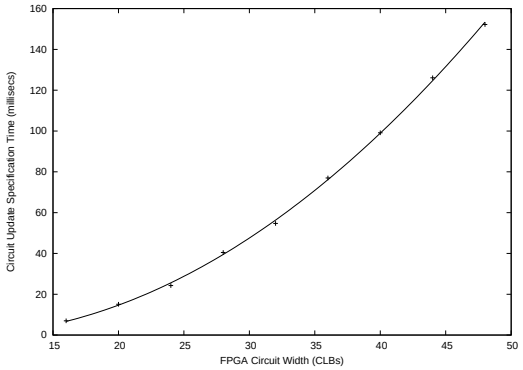
When the frontend timing results are compared with that of the backend, we find that the time to update a circuit specification and to partially reconfigure the chip is many orders of magnitude greater than the VHM time. This is because the VHM is much simpler as compared to the back end. The VHM only traverses a state-graph at run time. The backend, on the other hand, does bit manipulation in Java which is a costly operation.

The main advantage of the layout schema is that it eliminates the problems of re-routing between updates. These problems include configuration delay and the possibility of contention. The tradeoff is that the proportion of CLBs used in the computation is reduced by the fraction that are configured as constants to propagate partial result signals (minterms, requesters) across the FPGA. For example, in a requester module, only one CLB is used to evaluate whether the signal is in the required state for a transition, while the other CLBs are merely propagating this signal up the FPGA for the asserters.

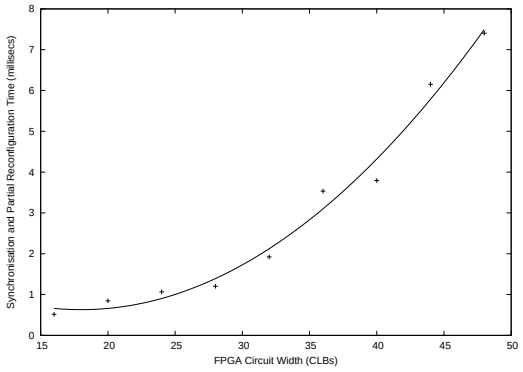
The column-based nature of configuration frames in the Virtex constrained the placement of circuit components for rapid reconfiguration. The ability to address and reconfigure individual LUTs using random addressing such as in the XC6200 allows more freedom for dynamic reconfigurable applications.



(a) Circuit Initialisation



(b) Circuit Update Specification



(c) Partial Reconfiguration

Figure 6. FPGA configuration run-times and circuit width

5. Conclusions and Future Work

We have developed an FPGA interpreter for high-level behavioural specifications given in the Circal process algebra. This interpreter allows circuits that do not statically fit into the available device area to be partitioned, mapped and loaded as execution proceeds.

We exploit the Circal characteristics that systems are composed of concurrently active processes and the temporal locality of activity within process task graphs in order to be able to effectively extend the design flow into the execution period of a computation.

Providing an abstraction layer for automated hardware virtualization for Circal designs goes a step towards increasing the reusability of the hardware and thus bringing programmable logic into mainstream computing. We believe that the concepts described in this paper can be applied to systems working in a remote environment where hardware upgrades are not that easy.

The performance analysis suggests that we have made some useful progress towards having an effective hardware management system for Circal designs. Not only does it remain to be seen how well these ideas translate to common design paradigms, we also have considerable work ahead to find more efficient ways of generating the partial bitstreams needed for this approach to be more generally useful. Nevertheless, the method is adequate for design space exploration when the developer is wanting to interact with a design prototype.

It is our goal to leverage these concepts to extend the semantic structures of Circal to have a simple yet powerful syntax for describing dynamic processes that can be mapped automatically to programmable logic devices. In order to be able to create and destroy processes at run time we will need to augment the concept of fixed routing structures (imposed upon us by the overheads inherent in the JBits router), with the ability to rapidly blank and/or reroute potentially large areas of a device. Perhaps to facilitate dynamic re-routing a specialised router that maintains a database of routing resources used needs to be developed.

While it is incumbent upon researchers and users to demonstrate the potential benefits of run time design elaboration (adapting to resource availability being one of them), we believe it would be of more general and long-term benefit were device vendors to open up their bitstream specifications to facilitate the development of tools that would allow the possibilities to be explored.

References

[1] P. Bellows and B. L. Hutchings. JHDL - an HDL for reconfigurable systems. In J. M. Arnold and K. L. Pocek, editors,

Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, pages 175–184, Napa, CA, Apr. 1998.

[2] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *ACM International Conference on Functional Programming*, 1998.

[3] G. Brebner. The swappable logic unit: a paradigm for virtual hardware. In K. L. Pocek and J. M. Arnold, editors, *The 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'97)*, pages 77 – 86, Los Alamitos, CA, Apr. 1997. IEEE Computer Society Press.

[4] Celoxica Ltd. *RC1000 Hardware Reference Manual*, 2001.

[5] O. Diessel and U. Malik. An FPGA interpreter with virtual hardware management. In *Proceedings, International Parallel and Distributed Processing Symposium, IPDPS 2002 Abstracts and CD-ROM*, page 155, Los Alamitos, CA, 2002. IEEE Computer Society Press.

[6] O. Diessel and G. Milne. A hardware compiler realizing concurrent processes in reconfigurable logic. *IEE Proceedings — Computers and Digital Techniques*, 148(4):152 – 162, Sept. 2001.

[7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science. Prentice-Hall International (UK), first edition, 1985.

[8] O. Mencer, M. Morf, and M. J. Flynn. PAM-Blox: High performance FPGA design for adaptive computing. In *Proceedings of the 6th Annual IEEE Symposium on Field Programmable Custom Computing Machines*, pages 167–174, Apr. 1998.

[9] G. Milne. CIRCAL and the representation of communication, concurrency and time. *ACM Transactions on Programming Languages and Systems*, 7(2):270–298, 1985.

[10] G. Milne. *Formal Specification and Verification of Digital Systems*. McGraw–Hill, London, UK, 1994.

[11] G. Milne. A model for dynamic adaptation in reconfigurable hardware systems. In A. Stoica, D. Keymeulen, and J. Lohn, editors, *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*, pages 161 – 169, Los Alamitos, CA, July 1999. IEEE Computer Society Press.

[12] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, first edition, 1980.

[13] I. Page. Constructing hardware–software systems from a single description. *Journal of VLSI Signal Processing*, 12(1):87 – 107, Jan. 1996.

[14] I. Page and W. Luk. Compiling Occam into FPGAs. In W. Moore and W. Luk, editors, *FPGAs: Proceedings of the International Workshop on Field Programmable Logic and Applications, (FPL '91)*, pages 271–283, Abingdon, Sept. 1991. Abingdon EE&CS Books.

[15] K. So. Compiling abstract behaviours to Field Programmable Gate Arrays. Undergraduate thesis, School of Computer Science and Engineering, University of New South Wales, Oct. 2001. Available at `http://www.cse.unsw.edu.au/~odiessel/papers/kso-ugthesis.ps.gz`.

[16] Xilinx, Inc. *Virtex 2.5V Field Programmable Gate Arrays*, Oct. 2000.

[17] Xilinx, Inc. *The JBits 2.8 SDK for Virtex*, Sept. 2001. In limited distribution release through Xilinx.