Compiling Process Algebraic Descriptions into Reconfigurable Logic

Oliver Diessel and George Milne

Advanced Computing Research Centre
School of Computer and Information Science
University of South Australia
Adelaide SA 5095
{Oliver.Diessel, George.Milne}@unisa.edu.au

Abstract. Reconfigurable computers based on field programmable gate array technology allow applications to be realized directly in digital logic. The inherent concurrency of hardware distinguishes such computers from microprocessor–based machines in which the concurrency of the underlying hardware is fixed and abstracted from the programmer by the software model. However, reconfigurable logic allows the potential to exploit "real" concurrency. We are therefore interested in knowing how to exploit this concurrency, how to model concurrent computations, and which languages allow us to control the hardware most effectively. The purpose of this paper is to demonstrate that behavioural descriptions expressed in a process algebraic language can be readily and intuitively compiled to reconfigurable logic and that this contributes to the goal of discovering appropriate high–level languages for run–time reconfiguration.

1 Introduction

The term *reconfigurable computer* is currently used to denote a machine based on field programmable gate array (FPGA) technology. This chip technology is programmable at the gate level thereby allowing any discrete digital logic system to be instantiated. It differs from the classical von Neumann computing paradigm in that a program does not reside in memory but rather an application is realized directly in digital logic.

For some computing and electronic control applications we are able to exploit the inherent concurrency of digital logic to directly realize algorithms as custom hardware to gain a performance advantage over software executing on conventional microprocessors. Given this observation, we may ask a wide range of questions, such as: how do we exploit this concurrency? How do we harness

it to perform computations? How do we model such computation? And what programming languages should we use to help programmers/designers?

This paper demonstrates that we can intuitively and rapidly compile a high–level language that is oriented to describing concurrency and communication into reconfigurable logic. We show how the core features of process algebras [5, 7, 4] and the Circal process algebra in particular [5, 6] can be mapped into reconfigurable logic.

The rationale for focusing on using a process algebra as the basis of a language for specifying reconfigurable logic are that it expresses the behaviour of a design in an abstract, technology–independent fashion and it emphasizes computation in terms of a hierarchical, modular, and interconnected structure. Process algebra have an extensive track record in the expression and representation of highly concurrent systems including digital hardware [1, 6] and are thus a good basis for a high–level language.

A high–level language based on process algebra is quite different from classical hardware description languages, such as VHDL and Verilog, that are oriented towards register–transfer and gate–level descriptions. Instead, this approach provides designers with a design paradigm focussed on behavioural process modules and their interconnection. Because of its modular focus, our approach aids the rapid compilation and partial reconfiguration of designs at run–time. Our approach also presents us with the potential for formally verifying the compilation algorithm. Related research on verifiable compilation to FPGAs was performed by Shaw and Milne [9] while Page and Luk [8] also constructed an Occam to FPGA compiler.

Circal models emphasize the control of and communication between processes. The rapid compilation of Circal models allows assemblies of interacting finite state machines to be implemented quickly. Apart from logic controllers, we may thus be able to build and quickly modify test pattern generators that function at near hardware speed. This project also aims to support dynamic structures that may facilitate the control of dynamically reconfigurable logic.

In the following section we provide an overview of the Circal process algebra and the source language for our compiler. Section 3 introduces our contribution with an overview of the compiler. We describe a technology–independent circuit model of Circal processes in Section 4. The mapping of these circuits to FPGAs, and Xilinx XC6200 chips in particular, is discussed in Section 5. The derivation of the mapping from behavioural Circal descriptions is outlined in Section 6. A summary of the paper and directions for further work are presented in Section 7.

2   The Circal process algebra

Circal is an event–based language; processes interact by participating in events, and sets of simultaneous events are termed actions. For an event to occur, all processes that include the event in their specification must be in a state that allows them to participate in the event. The Circal language primitives are:

State Definition $P \leftarrow Q$ defines process P to have the behaviour of term Q.

Termination $/\backslash$ is a deadlock state from which a process cannot evolve.

Guarding $a\,P$ is a process that synchronizes to perform event a and then behaves as P. $(a\,b)\,P$ synchronizes with events a and b simultaneously and then behaves as P.

Choice $P + Q$ is a term that chooses between the actions in process P and those in Q, the choice depending upon the environment in which the process is executed.

Non–determinism $P\,\&\,Q$ defines an internal choice that is determined by the process without influence from its environment.

Composition $P * Q$ runs P and Q in parallel, with synchronization occuring over similarly named events.

Abstraction $P - a$ hides event set a from P, the actions in a becoming unobservable.

Relabelling $P[a/b]$ replaces references to event b in P with the event named a.

3 Overview of compiler operation

This paper describes our efforts to implement a subset of Circal suited to the instantiation of Circal process models as reconfigurable logic circuits. The implementation of the hardware compiler is referred to as HCircal.

An HCircal source file consists of a declaration part, a process definition part, and an implementation part. Events and processes must be declared before use. The definition part consists of a sequence of process definitions adhering to the Circal BNF. The implementation part is introduced with the `Implement` declarative and is followed by a comma–delimited list of process compositions that is to be implemented in hardware. Processes must be defined before they are referred to in an `Implement` statement. HCircal does not currently allow the user to model non–determinism, abstraction, or relabelling. However, implementations of abstraction and relabelling are straightforward extensions to the current system.

In outline, the HCircal compiler operates as follows:

1. The user inputs an HCircal specification of the system to be implemented.
2. A compiler analyses the specification to produce a hardware implementation and a driver program for interacting with the hardware model
   - The current hardware model is in the form of a Xilinx XC6200 FPGA configuration bitstream [10] suitable for loading onto XC6200–based reconfigurable coprocessors such as the SPACE.2 board [3].
   - The driver program is a C program that executes on the host. The program loads the configuration onto the coprocessor and allows the user to interact with the implemented system.
3. The user runs the driver program and interacts with the hardware model by entering event traces and observing the system response.

The following sections describe the mapping from behavioural descriptions to technology–independent circuits, the decomposition of the circuits into modules

for which FPGA configurations are readily generated, and the derivation of the module parameters from the Circal specification. The generation of the host program is a straightforward specialization of a general program that obtains appropriate event inputs, loads the input registers, and reads the process state registers. It is not further discussed.

4   A circuit model of Circal

The aim of the model is to represent, as faithfully as possible, Circal semantics in hardware. The design concentrates on the representation of the Circal composition operator, which is of central importance because it is through the composition of processes that interesting behaviour is established. When processes are composed in hardware they are executed concurrently.

The hardware implementation of the Circal system follows design principles that aim to generate fast circuits quickly. The first of these is that, for the sake of speed and scalability, the hardware representation of Circal aims to minimize its dependence upon global computation at the compilation and execution phases. The second principle is that we choose to design for ease of run–time instantiation and computational speed over area minimality. The motivation for these choices is the desire to leverage the speedup afforded by concurrently executing the Circal system in hardware; they are supported by the ability to reconfigure the gate array at run–time in order to provide a limitless circuit area. Finally, we desire a reusable design because we believe that will facilitate design synthesis, circuit reconfiguration, and future investigations into dynamically structured Circal.

4.1   Design outline

A block diagram of a digital circuit that implements a composition of Circal processes in hardware. is shown in Figure 1(a).

The circuit consists of a set of interconnected processes that respond to inputs from the environment by undergoing state transitions. Processes are implemented as blocks of logic with state. In a given state, each process responds to events according to the Circal process definitions. Individual processes examine the event offered by the environment and produce a "request to synchronize" signal if the event is found to be acceptable. The request signals for all processes are then reduced to a single synchronization signal that each process responds to independently.

Implementing Circal in synchronous FPGA circuits leads us to assume that: an event occurs at most once during a clock period; the next state is determined by the events that occurred during the previous clock period; and, if no event occurs between consecutive positive clock edges, then the idling transition $P \rightarrow P$ occurs upon the second clock edge by default.
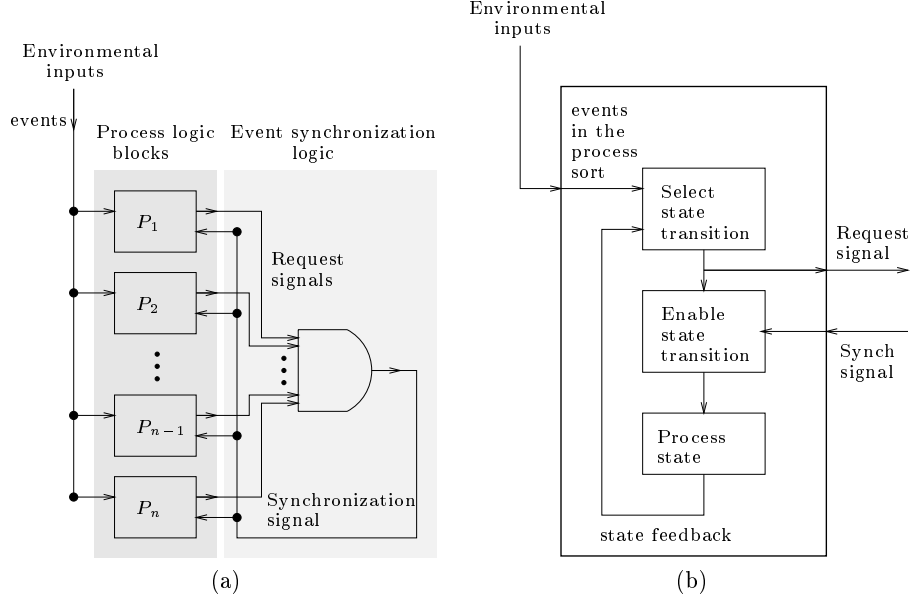
Fig. 1. (a) Circuit block diagram, and (b) Circal process logic block.

4.2 Process logic design

Process logic blocks are derived from the process definition syntax and represented as compact localized blocks of logic to simplify the placement and routing of the system. A high–level view of a process logic block is given in Figure 1(b).

A process is designed to respond to events in the environment that are acceptable to all processes in the composed system. In order to perform this function, the process logic first checks whether the event is acceptable to itself. If all processes find the event acceptable, the event synchronization logic returns a synchronization signal that is used by individual process logic blocks to enable the state transition guarded by the event. The following subsections describe the process logic design in more detail.

Determining the validity of event combinations We construct a combinational circuit that checks whether the events in the sort of the process form a valid guard for the current state. The process also accepts a null event (an event not in its sort) in order to allow other processes to respond to events it does not care about. The current state of the process is recycled if an unacceptable or null event is offered by the environment.

Let us assume at most k possibly recursive definitions $P_0, P_1, \ldots, P_{k-1}$ are necessary to describe the evolution of process P with sort $S = \{e_0, e_1, ..., e_{n-1}\}$, and that P_i, with $0 \leq i \leq k-1$, is defined as $P_i \leftarrow g_{i,0} P_{i,0} + ... + g_{i,j} P_{i,j} + ... + g_{i,j_i} P_{i,j_i}$, where index i refers to the current state, $P_{i,j}$ is the next state,

$P_0, ..., P_{k-1}$, the state P_i evolves to under guard $g_{i,j} \subseteq S$, and $g_{i,j}$ is interpreted as the simultaneous occurrence of the events in $g_{i,j}$. The definition for P_i consists of $j_i + 1$ guarded terms where the $g_{i,j}$ are all distinct. Note that there may be at most k distinct next states but $2^n - 1$ distinct guards.

If we think of the events and states as *boolean* variables, then in state P_i the process responds to event combinations in the set $\{\gamma_{i,j}\} \cup \{\nu_S\}$, where $\gamma_{i,j} = \varepsilon_0 \varepsilon_1 ... \varepsilon_{n-1}$ and $\varepsilon_l = e_l$ or $\varepsilon_l = \overline{e_l}$, for $0 \le l \le n-1$, depending upon whether or not $e_l \in g_{i,j}$, and where $\nu_S = \overline{e_0}\, \overline{e_1} ... \overline{e_{n-1}}$ is the null event for sort S. Process P in state P_i therefore accepts the boolean expression of events $\nu_S + \sum_{0 \le j \le j_i} \gamma_{i,j}$.

The request for synchronization signal, r_P, is thus formed from the expressions for all states: $r_P = \sum_{0 \le i \le k-1} (\nu_S + \sum_{0 \le j \le j_i} \gamma_{i,j}).P_i$.

Checking the acceptability of an event The request signals for all processes are ANDed together in an AND gate tree that is implemented external to the individual process logic blocks. The output of the tree is fed back to each process as the synchronization signal, s.

Enabling state transitions The state of the process is stored in flip–flops — one for each state. Let $D_{P_l}, 0 \le l \le k-1$, denote the boolean input function of the D–type flip–flop for state P_l. Then we can derive the following boolean equations from the process definitions: $D_{P_l} = s.(\nu_S.P_l + \sum_{0 \le i \le k-1} \sum_{P_{i,j} = P_l} \gamma_{i,j}.P_{i,j}) + \overline{s}.P_l$, for $0 \le l \le k - 1$.

In the above equations, the terms in parentheses are enabled when the synchronization signal, s, is high. These terms correspond to the guards on state transitions and to state recycling if a null event was offered to this process. The last term in the equations forces the current state to be renewed if the processes could not accept the event combination offered by the environment. By observing the synchronization signal, the environment can determine whether or not an event was accepted and can thus be constrained by the process composition.

4.3 The complete process logic block

The disjunction of the parenthesized terms in the flip–flop input functions implements the same boolean function as that to obtain the request signal. We therefore use the state selection circuits to form the request signal and use the synchronization signal to enable the selection.

5 Mapping circuits to reconfigurable logic

In this section we consider the placement and routing of the circuits derived in Section 4. The derivation of circuit requirements from the specification is discussed in the next section.

Our primary compilation goal is to generate FPGA configurations rapidly. We also want to be able to replace circuitry at run–time to explore changing process

behaviours and to overcome resource limitations. For this reason we're interested in mapping to Xilinx XC6200 technology because its open architecture allows us to produce our own tools and because the chip is partially reconfigurable.

Difficulties with placing and routing the Circal models satisfactorily with XACTStep, the Xilinx APR tool for XC6200, led us to consider decomposing the circuits into modules that can be placed and routed under program control. These modules serve as an attractive intermediate form since they are easily derived from the specification, they completely describe the circuits to be implemented in a hardware–independent manner, and the FPGA configuration can be generated without further analysis.

The circuits described in Section 4 are specified in terms of parameterised modules that communicate via adjoining ports when they are abutted on the array surface. To simplify the layout of the circuits, all modules are rectangular in shape. The internal layout of modules is also simplified by using local interconnects only. The module representation of the circuits is readily mapped to a particular hardware technology by suitable module generators. The compiler can thus be ported to a new FPGA type by implementing a new set of module generators.

We distinguish between 9 module types. Each module type implements a specific combinational logic function using a particular spatial arrangement. Modules are specified in terms of their location on the array, input and/or output wire bit vectors, and the specific function they are to implement, e.g., minterm number. The interested reader is referred to our technical report for a complete description of the module functions, parameters, and circuit generators [2].

6 Deriving modules from process descriptions

For each unique process that is to be implemented, a process template that consists of the modules comprising the process logic is constructed. The module parameters for a process template are independently calculated using relative offsets. Once the size of the logic for each template is known, a copy with absolute offsets (final placement of modules) is made for each process to be implemented. When all the parameters are known, the FPGA configuration is generated.

Currently the compilation is performed off–line and the configurations generated are static. In future implementations we plan to experiment with replacing modules at run–time to overcome resource limitations and implement dynamically changing process behaviours. Minor behavioural changes may simply involve replacing minterms or guard modules which could be done very quickly. The regular shapes and small sizes of modules may allow us to distribute them and finalize the module positioning at run–time in order to maximize array utilization.

For a more detailed description of the steps in the derivation of the module representation please refer to [2].

7   Conclusions

We have shown how to model Circal processes as circuits that can be mapped to blocks of logic on a reconfigurable chip. Modelling system components as independent blocks of logic allows them to be generated independently, to be implemented in a distributed fashion, to operate concurrently, and to be swapped to overcome resource limitations. The model thus exploits the hierarchy and modularity inherent in behavioural descriptions to support virtualization of hardware.

We have shown how to instantiate a circuit by decomposing it into parametric modules that perform functions above the gate level. To simplify the layout, modules are mapped to rectangular regions that are wired together by abutting them on a chip. Since the modules completely describe the circuits to be implemented in a hardware–independent yet readily mapped manner, they could serve as a mobile description of Circal processes that can be transmitted and instantiated remotely.

Future work will investigate developing an interpreter that adapts to resource availability and supports dynamic process behaviour. We also intend assessing the usability of process algebraic specifications for a number of applications. A further direction is to enhance the HCircal language to support stream–oriented and data–parallel computations.

Acknowledgements

References

1. A. Bailey, G. A. McCaskill, and G. Milne. An exercise in the automatic verification of asynchronous designs. *Formal Methods in System Design*, 4(3):213–242, 1994.
2. O. Diessel and G. Milne. Compiling HCircal. Draft manuscript, Advanced Computing Research Centre, University of South Australia, Adelaide, Australia, Spetember 24, 1999.
3. B. K. Gunther. SPACE 2 as a reconfigurable stream processor. In N. Sharda and A. Tam, editors, *Proceedings of PART'97 the 4th Australasian Conference on Parallel and Real–Time Systems*, pages 286 – 297, Singapore, Sept. 1997. Springer–Verlag.
4. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International series in computer science. Prentice–Hall, Inc., Englewood Cliffs, NJ, 1985.
5. G. Milne. CIRCAL and the representation of communication, concurrency and time. *ACM Transactions on Programming Languages and Systems*, 7(2):270–298, 1985.
6. G. Milne. *Formal Specification and Verification of Digital Systems*. McGraw–Hill, London, UK, 1994.
7. R. Milner. *Communication and Concurrency*. Prentice–Hall, Inc., New York, NY, 1989.

8. I. Page and W. Luk. Compiling Occam into FPGAs. In W. R. Moore and W. Luk, editors, *FPGAs, Edited from the Oxford 1991 International Workshop on Field Programmable Logic and Applications*, pages 271 – 283, Abingdon, England, 1991. Abingdon EE&CS Books.

9. P. Shaw and G. Milne. A highly parallel FPGA–based machine and its formal verification. In H. Grünbacher and R. W. Hartenstein, editors, *Second International Workshop on Field–Programmable Logic and Applications*, volume 705 of *Lecture Notes in Computer Science*, pages 162–173, Berlin, Germany, Sept. 1992. Springer–Verlag.

10. Xilinx. *XC6200 Field Programmable Gate Arrays*. Xilinx, Inc., Apr. 1997.