Behavioural Language Compilation with Virtual Hardware Management

Oliver Diessel[1] and George Milne[2]

[1] School of Computer Science and Engineering
University of New South Wales, Australia
odiessel@cse.unsw.edu.au
[2] Advanced Computing Research Centre
School of Computer and Information Science
University of South Australia, Australia
milne@cis.unisa.edu.au

Abstract. High-level, behavioural language specification is seen as a significant strategy for overcoming the complexity of designing useful and interesting reconfigurable computing applications. However, appropriate frameworks for the design of behaviourally specified systems are still being sought. We are investigating behavioural language and compiler design based on the Circal process algebra, which is a natural framework within which to describe the concurrent activity of reconfigurable logic circuits. In this paper we describe an FPGA interpreter that exploits the inherent concurrency, hierarchy, and modularity of Circal and its circuit realization to automatically manage hardware virtualization. The techniques employed by the interpreter may be used to overcome resource limitations and adapt circuits to changing application needs at run time.

1 Introduction

One of the attractions of reconfigurable computing systems based on field programmable gate array (FPGA) technology is that the circuit realizing a design may change during the run time of an application to achieve better performance, to reflect changes in the application model, or to make better use of the underlying hardware.

This capability has been difficult to realize in practice due to: a lack of suitable languages in which to describe the attributes of applications that can profit from and exploit reconfigurable logic; the complexity of the steps that need to be taken in designing reconfigurable computing applications; and the

This paper appears in Reiner W. Hartenstein et al., editors, *Field-Programmable Logic and Applications, The Roadmap to Reconfigurable Computing, 10th International Workshop, FPL 2000 Proceedings*, volume 1896 of Lecture Notes in Computer Science, pp. 707 – 717, Berlin, Germany, 2000. Springer-Verlag.

perception that users need to be highly skilled in both software and hardware engineering to design applications that achieve performance or cost benefits over application specific integrated circuits and/or microprocessors. For these reasons, reconfigurable technology has not attracted systems designers who would like to program reconfigurable systems in an abstract, technology-independent manner using high-level language concepts.

Traditionally, the specification of reconfigurable computing applications has been attempted with the aid of inherently structural hardware description languages (HDLs) such as VHDL [2] and Verilog [18], (augmented) programming languages such as C++ [1], and schematic design entry, none of which are ideal since they take a low-level view of systems and their implementation. It is therefore difficult to describe systems at a more abstract behavioural level as we do with high-level programming languages that are oriented towards the function of a program rather than its realization on the underlying hardware.

To bring reconfigurable computing to the mainstream, to make it more accessible, and to permit faster prototype turnaround, we shall need to allow design entry via suitable high level languages (HLLs). Such languages will allow people more versed in algorithms and applications and less experienced with the underlying hardware details to apply and experiment with reconfigurable computing.

We have adopted the Circal [9] process algebra as the basis of such a language for a number of reasons. First, process algebras (PAs) such as Circal, CSP [7], and CCS [13] are formal languages developed for the purpose of describing concurrent systems. We believe them to be appropriate for describing FPGAs since, as with all VLSI digital logic, they are designed to represent highly concurrent systems. As such, there is a good match between language concepts and hardware realization vis-a-vis the expression of parallelism and the inherent parallelism of digital logic. Second, PAs are elegant, simple, yet powerful formalisms in which it is easier to explore fundamental language issues than with HDLs and programming languages. Third, traditional applications of reconfigurable logic such as prototyping and system control, even the design of the control part of data-oriented applications, are modelled as interacting finite state machines (FSMs) — PAs have essentially been created for the purpose of describing assemblies of interacting FSMs at a behavioural level. Fourth, there is the hope that a top-down, hierarchical, and modular focus, as emphasized by a PA such as Circal, will aid synthesis because ever more complex structures may be built through assembly, while the effort required to design each module (the unit of design) remains relatively constant.

This research is related to earlier work in compiling Occam to FPGAs [15, 17, 8, 16]. The Occam language was seen as a natural language for design input due to its simplicity and its facility for expressing parallelism. There was also a belief that the constructive approach used to translate Occam into digital logic and the formal semantics of the language could lead to the development of a verifiably correct hardware compiler.

The results reported here extend this research direction and investigate how to express and control dynamic reconfiguration through the use of a language

that supports the description of changing hardware structures, such as occurs during reconfiguration, to overcome resource limitations or to adapt circuits to the changing needs of an application. This paper reports on the techniques we have developed to overcome resource limitations and applied to an interpreter for such a language, based on the Circal process algebra. We believe the ideas presented here will form the basis for the design of a language for reconfigurable computing modelled on a process algebra that supports the direct description of dynamically changing hardware structure [11].

In [5] we described an FPGA compiler for Circal that synthesizes complex systems from high-level behavioural descriptions. In this paper, we describe new features of a Circal interpreter that exploits language features in order to reconfigure hardware to overcome resource limitations. This scheme allows the necessary circuitry to be loaded as dictated by execution flow at run time.

2 Description of the Circal language

This section presents Circal as a descriptive medium for reconfigurable computing. We describe the key language concepts and how they are used to describe concurrent systems. We also introduce a simple example that serves to illustrate these concepts, the presentation of the Circal translation scheme, and our hardware management strategies later in the paper.

Circal is a formal language used to model the behaviour of complex, concurrent systems in a constructive, modular, and hierarchical manner by (1) modelling the behaviour of its component processes, and (2) by modelling the interaction of these component processes in terms of how they communicate events or actions between themselves. Systems, and the processes that model their behaviour, are described hierarchically and in a modular fashion. The description of a system thus typically proceeds in a top-down manner with the elaboration of component processes leading to further decomposition until the desired level of description is reached [10].

Circal is an event–based language and processes interact by participating in the occurrence of events. 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. Process Q is given the name P.

Termination Δ 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, or evolves to, 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. Usually the choice is mediated through the offering by the environment of a guarding event.

Non–determinism $P \,\&\, Q$ defines an internal choice that is determined by the process without influence from its environment. Either branch might be taken by the process, the reason for the choice being unobservable.

Composition $P * Q$ runs P and Q in parallel, with synchronization occurring over similarly named events. When P and Q share a common event, both must be in a state in which they can accept that event before the event and synchronous state evolution can occur. P and Q may independently respond to events that are unique to their specification. Should such independent events occur simultaneously, the processes respond simultaneously.

Abstraction $P - a$ hides event set a from P, the actions in a becoming encapsulated and unobservable. Unobservable events internal to a process lead to non-deterministic behaviour.

Relabelling $P[a/b]$ replaces references to event b in P with the event named a. This feature is similar to calling procedures with parameter substitution.

Circal differs from most Process Algebras in that it has a strict interpretation of the response of processes to the simultaneous occurrence of events and is therefore well-suited to modelling synchronous devices such as FPGAs.

Circal has been used extensively to describe systems composed of interacting finite state machines, to describe control paths for digital systems, to describe asynchronous logic and to specify cellular automata for FPGA implementation [10]. To demonstrate the approach to modelling adopted in a Circal descriptive framework, consider a mobile phone system that is also capable of receiving and recording television or radio broadcast signals. We shall examine the description and composition of parts of the broadcast receiver subsystem B and the phone subsystem P. Consider a mode of operation in which the user initiates broadcast reception by the s action used to select a channel. Should an urgent phone call arrive, the system is able to buffer reception of the broadcast while the user answers the phone with action a. When the user hangs the phone up, the broadcast may be resumed from the time it was interrupted, by use of event r, and the remainder of the reception is buffered until the user flushes the buffer by selecting a new channel or terminates reception with another r event.

In this presentation, we depict a block diagram of the structure of the composed subsystems of the mobile phone M in Figure 1. The following Circal

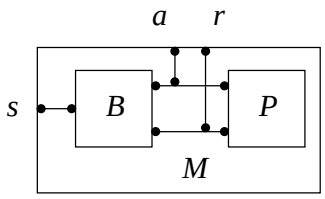

Fig. 1. Mobile phone system comprising broadcast and phone subsystems.

definitions describe the broadcast component B in terms of the four distinct

states it may take.

$$B_i \leftarrow s\,B_r + a\,B_i + r\,B_i, \text{ when } B \text{ is inactive,} \quad (1)$$
$$B_r \leftarrow a\,B_s + s\,B_r + r\,B_i, \text{ when } B \text{ is receiving,} \quad (2)$$
$$B_s \leftarrow r\,B_b, \text{ when } B \text{ is storing, and} \quad (3)$$
$$B_b \leftarrow s\,B_r + r\,B_i, \text{ when } B \text{ is buffering.} \quad (4)$$

The phone component may be defined as

$$P_i \leftarrow a\,P_a + r\,P_i, \text{ when } P \text{ is inactive, and}$$
$$P_a \leftarrow r\,P_i, \text{ when } P \text{ is answering.}$$

Finally, we define the mobile phone system M as the composition or synthesis of the two components B and P by $M \leftarrow B * P$.

Such Circal expressions define the behaviour when in a given state in terms of the occurrence of permitted events, that in this example model the interaction between the system and the user.

The focus of this paper is on describing and managing finite FPGA resources to create a larger virtual resource using Circal as the descriptive medium. In terms of the example of Figure 1, we provide techniques for automatically swapping active subcomponents of M into hardware when there is insufficient resource to implement all components of the system at once.

3 Description of the compiler

In [5] we described a compiler that derives and implements a digital logic representation of high-level behavioural descriptions of systems specified using Circal.

Significantly, this compiler structures the derived circuits so as to reflect the design hierarchy and interconnection of process modules given in the specification. This approach simplifies the composition of modules since the majority of interconnections that are to be implemented are between co-located blocks of logic and the replacement or exchange of system modules is facilitated by the replacement of a compact region rather than of distributed logic. At the topmost design level, the circuit is clustered into blocks of logic that correspond to the processes of a system. These are wired together on similarly labelled ports to effect event broadcast and to allow process state transitions to be synchronized. Our strategy for virtual hardware management makes use of an interpreter that follows a similar translation philosophy.

An overview of the realization of Circal expressions in digital logic is depicted in Figure 2(a). The process logic blocks individually implement circuits with behaviours corresponding to the component processes of the specification — see Figure 2(b). Each block is provided with inputs corresponding to the events in its sort. Events are realized by the presence or absence of signals that are generated by the environment on similarly named wires. The response of process logic blocks to an event is determined by the global acceptability of an event.

Processes independently assert a request signal when acceptable events for the current state are offered by the environment. Synchronized state evolution occurs upon the next clock edge if all processes agree on the acceptability of the event.

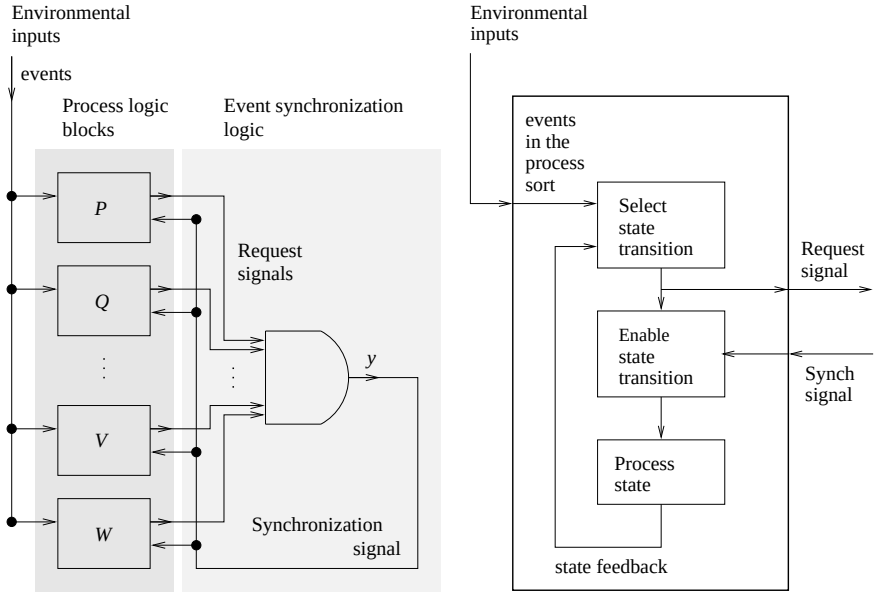

Fig. 2. (a) Circuit block diagram, and (b) Circal process logic block.

Below the process level in the hierarchy, the circuits are partitioned into component circuit modules that implement combinational logic functions of minor complexity. A typical example of such a module generates a minterm that recognizes an acceptable event combination; another forms the disjunction of several such minterms in order to recognize those event combinations that are acceptable in a particular state. Modules are rectangular in shape and are laid out onto abutting regions of the array surface. Signals flow from one module to another via aligned ports. The partitioning of the system circuit into component circuit modules is fixed during the analysis phase of the compiler, which is programmed with a particular arrangement of the modules in mind — the module types and their relative position is thus directed by the compilation process [6].

The component circuit modules are each specified by a number of parameters, the derivation of which represents the goal of the analysis phase of the compiler. The following synthesis phase applies each set of module parameters to a corresponding module generator that maps its functional and spatial requirements to FPGA resources and produces a bitstream fragment for the circuit component. Circuit module generators perform the physical mapping of the circuit to FPGA resources in order to obtain quick physical designs and to control the layout.

These factors simplify reconfiguration by allowing precise changes to be carried out quickly.

For example, to implement the process logic for the broadcast component B, the Circal definitions (1) through (4) in Section 2 are translated into a set of boolean equations over similarly labelled literals. One equation describes the logic for the request signal r_B that should be generated for the block,

$$r_B = (s.\overline{a}.\overline{r} + \overline{s}.a.\overline{r} + \overline{s}.\overline{a}.r + \overline{s}.\overline{a}.\overline{r}).B_i + (\overline{s}.a.\overline{r} + s.\overline{a}.\overline{r} + \overline{s}.\overline{a}.r + \overline{s}.\overline{a}.\overline{r}).B_r$$
$$+(\overline{s}.\overline{a}.r + \overline{s}.\overline{a}.\overline{r}).B_s + (s.\overline{a}.\overline{r} + \overline{s}.\overline{a}.r + \overline{s}.\overline{a}.\overline{r}).B_b$$

The rest describe the input functions for the (D-type) flip-flops implementing the 4 states of the process within the "Enable state transition" block of Figure 2(b). These equations include a literal for the synchronization signal y.

$$D_{B_i} = y.([\overline{s}.a.\overline{r} + \overline{s}.\overline{a}.r + \overline{s}.\overline{a}.\overline{r}].B_i + \overline{s}.\overline{a}.r.B_r + \overline{s}.\overline{a}.r.B_b) + \overline{y}.B_i$$
$$D_{B_r} = y.(s.\overline{a}.\overline{r}.B_i + [s.\overline{a}.\overline{r} + \overline{s}.\overline{a}.\overline{r}].B_r + s.\overline{a}.\overline{r}.B_b) + \overline{y}.B_r$$
$$D_{B_s} = y.(\overline{s}.a.\overline{r}.B_r + \overline{s}.\overline{a}.\overline{r}.B_s) + \overline{y}.B_s$$
$$D_{B_b} = y.(\overline{s}.\overline{a}.r.B_s + \overline{s}.\overline{a}.\overline{r}.B_b) + \overline{y}.B_b$$

Note that most of the terms in these equations are formed from the conjunction of a minterm defined over the events the process can respond to and a process state. In fact the terms in r_B are covered by the parenthesized terms of the flip-flop input functions. Thus r_b is implemented as the disjunction of these latter terms. Each minterm, the disjunction of minterms that combine with a state, and the disjunction of terms that combine with y in a flip-flop input function, as well as the disjunction of terms to form r_B are encoded as parameterized modules and implemented as rectangular circuit fragments within the logic block for a process.

4 A Circal interpreter

The Circal compiler described above, and in more detail in [5, 6], produces a static design prior to circuit loading and execution. Using the same philosophy for realizing Circal in FPGA logic, the Circal interpreter described in this paper is, in contrast, able to finalize physical designs and modify designs at run time either in order to better manage limited reconfigurable resources or to realize dynamically reconfigurable systems specified in dynamically structured Circal (dsCircal), a development of Circal that permits the direct description of systems whose structure changes through time [11, 12].

There are several reasons for using an interpreter rather than a compiler to manage the implementation of a Circal design. First, we could compile and temporally partition the system off-line, but we may not know in advance of running the system which Circal processes need to be simultaneously active. If the partitioning is unsuitable, the performance of the system will be affected. Second, we need to be prepared to adapt to time-varying resource availability

such as in distributed (networked) or multitasking reconfigurable computing environments. A static design may be held up or may hold up other applications due to allocation conflicts. Third, we cannot use a static compilation approach if the behaviour and/or circuit structure is permitted to change over time, as may be described in the dsCircal language.

The Circal interpreter pre-processes a system description until functional module parameters such as the number of variables in minterm blocks are known. Some spatial parameters, such as sizes of modules and relative offsets are also determined at the pre-processing stage. At run time, the interpreter looks after loading modules on an as needs basis, employing techniques for managing the reconfigurable resource as described in Section 5. This involves finalizing module locations on the underlying FPGA substrate and generating the bitstream fragments for the modules as they are to be loaded. Bitstream generation at run time does not represent a significant overhead because the time to generate and emit the bitstream is proportional to its size. The interpreter suspends the application clock for the duration of the reconfiguration.

5 A strategy for virtual hardware management

The implementation of a system described in Circal exhibits both coarse and fine grained parallelism. A process logic block is a relatively large unit of computation or circuit that can be viewed as a "grain of large size", while the circuit module subcomponents of a process are small units of computation, typically no larger than a few gates, and may therefore be viewed as being of a much finer grain size. Realistic systems will be composed of multiple processes, while within individual processes high degrees of parallelism may exist between small subcomponents.

We exploit the two extremes of grain size to manage hardware use in three ways:

1. We sequentialize deployment and execution of circuitry by repeated reconfiguration when streams could be executed concurrently but there is insufficient resource to do so. This strategy is applied at the coarse grain size of individual processes.

2. We deploy circuit modules as signals advance through the circuit so as to allow concurrent execution yet reduce demand on resource. This strategy is intended to be employed at a fine grain size on time-multiplexed chips [4, 19] and relies upon allocating the subcomponents of processes to successive context layers so as to keep the cost of reconfiguration low.

3. We load logic as processes evolve when the choice of logic to be executed is determined at run time. This strategy is applied at both the coarse process and fine module grain sizes. On the one hand, the deployment of processes or assemblies of process logic may be guarded by events that will control the loading of circuitry should such an event occur. On the other hand, the need for certain modules, such as those to detect specific event combinations,

only arises in particular states. To save on resources, we thus only implement those modules that are needed given the current state of the system.

Apart from allowing large systems to run on limited FPGA resources, the benefit of dynamic circuit loading is that it saves loading unnecessary circuitry, which saves on load time and lowers demand for the resource. However, such a scheme requires very efficient techniques for loading circuitry to minimize reconfiguration overheads. We tackle this problem by primarily relying upon fast module generators that produce circuitry in time proportional to the time needed to load it. The overheads of the interpreter are thus kept low.

To illustrate the use of these techniques, we contrast the runtime control of the reconfigurable resource performed by the compiler with that carried out by the interpreter. The host program of the compiler that loads and runs the fixed, static circuits executes the following loop:

```
repeat {
  wait until event occurs
  present events to system
  allow system to determine its response and evolve state
} until system halts
```

The interpreter takes one of two alternative approaches depending upon whether the design, as described in Circal, is static or dynamic. If the design is static, it can be partitioned before execution, and all we need do at run time is cycle between the partitions. Thus the above loop becomes:

```
partition system by packing components into available area
repeat {
  wait until event occurs
  for each partition and while event still acceptable
    load partition and determine acceptability of event
  if event acceptable to entire system
    for each partition
      load partition and allow state to evolve
} until system halts
```

If on the other hand the design or the amount of available resource is dynamic, or we wish to employ the third strategy above, then partitioning is done at run time during the execution phase. The loop then becomes:

```
repeat {
  wait until event occurs
  while event acceptable and components remain to be processed
    compute partition by packing components into available area
    load partition and determine acceptability of event
  if event acceptable to system as a whole
    for each partition
      load and evolve state
} until system halts
```

In the mobile phone example pictured in Figure 1, these techniques would be employed if the circuit to implement M were too large, but B and P on their own fit into the available FPGA resource.

When a process can assume one of many states, as for process B in our example, rather than implementing the logic for all possible states in a single block, strategy 3 suggests we just implement the logic corresponding to the definition of the current state. As the state changes, the behaviour of the new state is implemented by reconfiguring the subcomponents that recognize acceptable events for this new state. We thus initially implement the boolean equations corresponding to the definition for state B_i,

$$B_i \leftarrow s\, B_r + a\, B_i + r\, B_i,$$

that is,

$$r_{B_i} = (s.\overline{a}.\overline{r} + \overline{s}.a.\overline{r} + \overline{s}.\overline{a}.r + \overline{s}.\overline{a}.\overline{r}),$$

and

$$D_{B_i} = y.(\overline{s}.a.\overline{r} + \overline{s}.\overline{a}.r + \overline{s}.\overline{a}.\overline{r}).B_i + \overline{y}.B_i, \text{ and}$$
$$D_{B_r} = y.s.\overline{a}.\overline{r}.B_i\ .$$

Should an s event occur, for example, the logic for B would be reconfigured to implement equations corresponding to the definition for state B_r.

6 Conclusions

In this paper, we have described a behavioural specification language for reconfigurable computing together with techniques for managing hardware use at run time by exploiting the hierarchy and modularity in the language and its implementation using FPGA technology. The techniques used have been described in the context of overcoming resource limitations. However, they are equally applicable to the implementation of systems whose circuit structure changes during execution.

Current work is being carried out to complete the implementation of the Circal interpreter and the testing of its performance. We are also investigating techniques and language constructs appropriate for the description and management of run-time, dynamically changing circuit structures such as described in [12].

Acknowledgement

This research was funded by the Australian Research Council.

References

1. P. Bertin and H. Touati. PAM programming environments: Practice and experience. In Buell and Pocek [3], pages 133 – 138.
2. D. C. Blight and R. D. McLeod. VHDL for FPGA design. In Moore and Luk [14], pages 246 – 254.
3. D. A. Buell and K. L. Pocek, editors. *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'94)*, Los Alamitos, CA, Apr. 1994. IEEE Computer Society Press.
4. A. DeHon. DPGA–coupled microprocessors: Commodity ICs for the early 21st Century. In Buell and Pocek [3], pages 31 – 39.
5. O. Diessel and G. Milne. Compiling process algebraic descriptions into reconfigurable logic. In J. Rolim, editor, *Parallel and Distributed Processing, 15 IPDPS 2000 Workshops Proceedings*, volume 1800 of *Lecture Notes in Computer Science*, pages 916 – 923, Berlin, Germany, 2000. Springer–Verlag.
6. O. Diessel and G. Milne. A hardware implementation of the Circal process algebra *and* compiling HCircal. Technical report ACRC–00–013, Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, Mawson Lakes, SA, 2000.
7. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International series in computer science. Prentice–Hall, Inc., Englewood Cliffs, NJ, 1985.
8. H. Jifeng, I. Page, and J. Bowen. Towards a provably correct hardware implementation of Occam. In G. J. Milne and L. Pierre, editors, *Correct Hardware Design and Verification Methods, IFIPWG10.2 Advanced Research Working Conference, CHARME'93 Proceedings*, volume 683 of *Lecture Notes in Computer Science*, pages 214 – 225, Berlin, Germany, May 1993. Springer–Verlag.
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. G. Milne. Modelling dynamic structures. Technical report ACRC–99–01, Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, Adelaide, Australia, Jan. 1999.
13. R. Milner. *Communication and Concurrency.* Prentice–Hall, Inc., New York, NY, 1989.
14. W. R. Moore and W. Luk, editors. *FPGAs, Edited from the Oxford 1991 International Workshop on Field Programmable Logic and Applications*, Abingdon, England, 1991. Abingdon EE&CS Books.
15. I. Page and W. Luk. Compiling Occam into FPGAs. In Moore and Luk [14], pages 271 – 283.
16. P. Shaw. *A Generic Approach to Compiling Occam into Circuits.* PhD thesis, Department of Computer Science, University of Strathclyde, Dec. 1994.
17. P. Shaw and G. Milne. A highly parallel FPGA–based machine and its formal verification. In H. Grünbacher and R. W. Hartenstein, editors, *Field–Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping, 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.

18. D. Soderman and Y. Panchul. Implementing C algorithms in reconfigurable hardware using *c2verilog*. In K. L. Pocek and J. M. Arnold, editors, *The 6th Annual IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'98)*, pages 339 – 342, Los Alamitos, CA, Apr. 1998. IEEE Computer Society Press.
19. S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time–multiplexed FPGA. In K. L. Pocek and J. M. Arnold, editors, *The 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'97)*, pages 22 – 28, Los Alamitos, CA, Apr. 1997. IEEE Computer Society Press.