An FPGA Interpreter with Virtual Hardware Management

Oliver Diessel Usama Malik
School of Computer Science and Engineering
University of New South Wales
Sydney NSW 2052
{odiessel,umalik}@cse.unsw.edu.au

Abstract

This paper describes the design of an interpreter that overcomes FPGA resource limitations for a class of control-oriented circuits by automatically partitioning, elaborating, and loading circuit components as directed by their execution. By providing a virtual hardware management facility, this enables us to implement large systems, specified in Circal, on small FPGA chips.

1. Introduction

Reconfigurable computers, based on FPGAs, allow applications to be realized directly in hardware. This has the potential to achieve high performance at a reasonable cost. However, the ready uptake and use of this technology continues to be hampered by the lack of convenient languages, tools, and abstractions

We are investigating the use of the Circal process algebra as the basis for high-level programming constructs and compilation techniques that can make use of the capabilities of the technology, including large-scale, fine-grained parallelism and adaptation through run-time reconfigurability. Circal allows the behaviour of control-oriented designs such as assemblies of interacting finite state machines to be captured in a very natural, abstract, incremental manner.

Previous work has resulted in a compiler that translates Circal specifications into FPGA circuits that might be used as controllers in embedded applications, or that can be interfaced with external sensors and actuators in their environment to perform control-oriented tasks [3]. The compiler produces a static circuit design that is constrained in its size and complexity by the available FPGA area. Automated mappings contribute to the inefficiency in the use of this area. However, the compiler was originally envisaged to be extended to support some form of circuit swapping. Options for doing so were explored in [2].

In this paper we outline our design for an interpreter that manages the run-time progress of a Circal computation, and that elaborates the design of and loads circuit components as directed by execution flow. Conceivably, this capability will benefit systems in which the available reconfigurable resource cannot be arbitrarily increased to suit the needs of applications. Satellite and consumer electronic devices are examples of systems for which the desired applications (and their size) continue to evolve after system deployment.

Previous efforts at implementing hardware virtualization have largely targeted data-oriented applications and have relied upon manual circuit partitioning and the design of specialized controllers that manage the swapping of circuit modules at run time [4, 5, 6, for example]. At present, tool support for hardware swapping is not readily available. Concepts for general, coarse-grained approaches to virtual hardware management have also been explored by several researchers [7, 1].

Our approach is necessarily different. Since we have as an ultimate goal high-level programming environments and support for reconfigurable computing, we aim to provide automated techniques for partitioning and run-time management. As much as possible, we aim to hide the details of providing these facilities from the user. At this stage of the development, there are significant costs associated with circuit switching and inefficient mappings, but we believe the progress so far is an exciting step towards the design and implementation of future systems that will provide such capabilities as a matter of course.

The regularity and modularity inherent in the Circal application specification and solution space allows us to investigate a consistent approach and to propose a fine-grained solution for a class of problems and their associated hardware realizations. We hope this work will provide us with insights that will lead to a more general solution to virtual hardware management that is applicable to data-oriented applications as well.

Section 2 describes Circal and the existing compiler. Section 3 describes the design of the interpreter. We con-

clude the paper with a brief summary of the results and outline the directions that we are interested in developing this work.

2. Circal compiler

Circal is a type of formal language, known as a process algebra, that is suited to modelling concurrent systems. Circal describes *systems* as compositions of interacting *processes* that synchronize on shared *events*. The behaviours of the processes are described independently and using a formal notation that describes their state evolution as events occur. When processes are composed together, the behaviour of the composed system is determined by process algebraic laws.

Process algebras such as Circal have traditionally been used for modelling, analyzing and specifying concurrent systems. We're interested in Circal's use as a descriptive medium for the exploration of high-level specification and compilation tools for dynamically reconfigurable systems. Simple yet powerful abstractions such as Circal are attractive for this purpose because they capture the essence of concurrent and synchronous behaviour without the clutter of sequential high-level languages, and they also offer the possibility of verifying the correctness of an implementation

2.1. Circal hardware description language

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 relevant to hardware design 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.

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.

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.

Since processes essentially describe finite state machine behaviours, Circal in its hardware descriptive form can be used to specify interacting finite state machines - instead of building a monolithic controller, using Circal we can specify its component behaviours and the composed behaviour can then be synthesized using a hardware compiler. Another current research thrust is directed at extending the descriptive capabilities of Circal to data-oriented applications

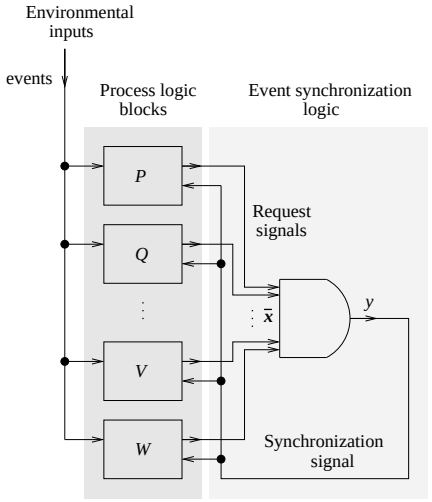
2.2. Overview of the XC6200 compiler

In [3] we described a compiler for the Xilinx XC6200 FPGA 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.

An overview of the realization of Circal expressions in digital logic is depicted in Figure 1(a). The process logic blocks individually implement circuits with behaviours corresponding to the component processes of the specification — see Figure 1(b). Each block is provided with inputs corresponding to the events in its *sort* (set of event ports). 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 accept-

able 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.



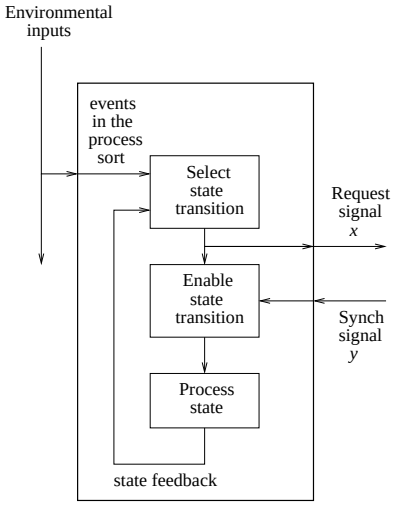


Figure 1. (a) Circuit block diagram, and (b) Circal process logic block.

2.3. Module representation of circuits

Below the process level in the hierarchy, the circuits are partitioned into component circuit modules that implement combinational logic functions of minor complexity. The module arrangement for a typical system $P * Q * \ldots$ is depicted in Figure 2.

We distinguish between 10 module types. Inputs are captured by an Environmental Inputs (EI) block and are routed to process logic by Bus (B) and Input Junction (IJ) blocks.

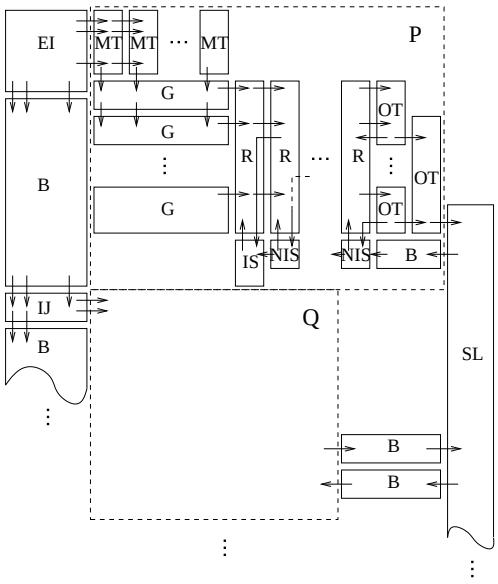


Figure 2. Typical circuit module arrangement

Within a process, a series of Minterm (MT) blocks detects the combinations of event inputs a process can respond to. The sets of minterms that lead to particular next states from a given current state are summed in so-called Guard (G) blocks. Associated with each Initial State (IS) and Non-Initial State (NIS) block is a Request (R) block that determines whether any of the Guard block outputs are accepted in the current state. The process request signals are formed from the disjunction of Request block outputs in OR Tree (OT) blocks, and the Synchronization Logic (SL) blocks form the synchronization signal from the individual process request signals.

Each module type implements a particular combinational logic function using a specific spatial arrangement. Modules are specified by giving the exact function to be implemented, e.g., a Minterm block is specified by the process sort size, the minterm number, and its location on the array. To simplify the layout of the circuits, all modules are rectangular in shape and communicate via adjoining ports when they are abutted on the array surface.

As an example, consider the process P defined by the following equations:

$$P \leftarrow P1 \quad (1)$$
$$P1 \leftarrow (a\,c)\,P2 + b\,P3 \quad (2)$$
$$P2 \leftarrow b\,P2 + a\,P3 \quad (3)$$
$$P3 \leftarrow (a\,b)\,P4 \quad (4)$$
$$P4 \leftarrow c\,P4 + a\,P2 \quad (5)$$

This process has the module representation of Figure 3. Event signals flow into the circuit at its top left edge. Each

minterm recognizes one of the event combinations, $(a\,b)$, b, etc. guarding a state transition. Event combinations that lead to identical transitions are combined below in guard blocks that have been labelled here with the minterm combination that is output and the transition it guards. Note the guard blocks are arranged in next state, current state order.

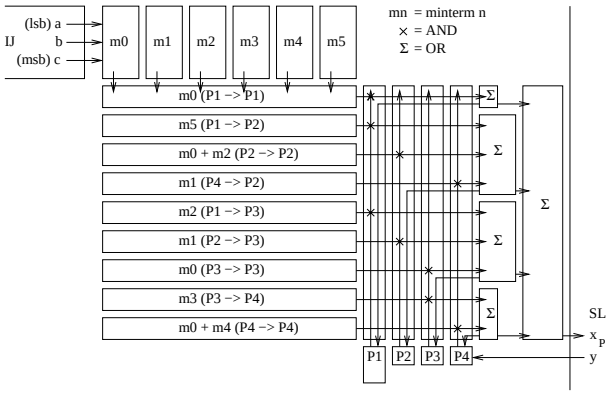


Figure 3. Module arrangement for process P

Circuits recycle the current state when the current event combination is not in the process sort or when the system cannot accept the current event. In the first case, none of the event signals for a process is asserted, which results in the *null event* for a process (minterm 0) being asserted. In the latter case, the synchronization signal is not asserted because some process does not accept the input event and hence does not assert its request signal.

An input event is acceptable if it guards a transition from the current state. The circuit performs this check by combining a guard block's output with its current state in a requester block. Those outputs that lead to the same next state are further combined to form a next state select signal that is fed back through the requesters to the appropriate state block. State is stored using a one-hot encoding. If a select signal is asserted, then the request signal, x_P, is also asserted. The synchronization signal, y, if asserted by the synchronization logic, then enables the transition to the next state that has been selected.

2.4. Compiler operation

Figure 4 provides an overview of the compiler's operations. A Circal specification is parsed and analyzed by the so-called front-end, which determines the logic required to implement the specification and produces the parameters for the set of modules required to implement the system. The set of module parameters is largely determined by the logical requirements of the circuits which are independent of the physical requirements of its implementation. A system configuration is then produced by generating a bitstream fragment for each module in the so-called back-end of the compiler. The host loads this configuration onto the coprocessor and controls interaction with the system. The user/environment provides event traces and observes/responds to the resulting changes in system state.

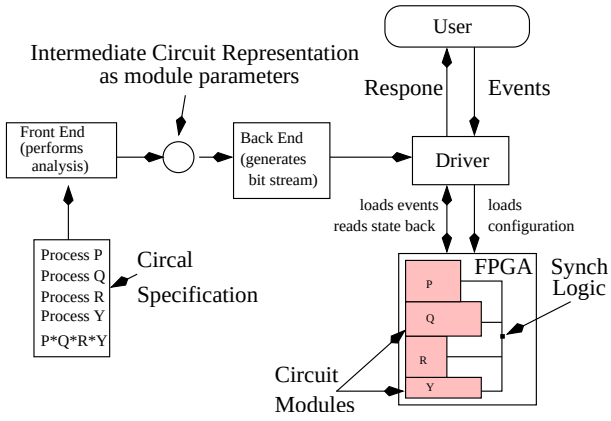


Figure 4. Overview of the compiler operation

3. Circal interpreter

3.1. Overview

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 circuit as they are needed.

This interpreter is constructed by inserting a virtual hardware manager (VHM) between the compiler front-end and the module generators of the compiler. The VHM determines through feedback from application execution which components to implement next.

Since the interpreter only knows at run time which components to load, the physical details of the circuits are finalized at run time. There is thus an ongoing interaction between the progress of computation and the operation of the design suite.

The design flow incorporates the function of run-time management and the ongoing configuration of the FPGA. Circuit design therefore does not complete until execution has finished.

3.2. Interpreter operation

The interpreter works by partitioning an intermediate representation of the circuits into functional (or more precisely, behavioural) components that are implemented as

they are needed. In order to manage the resource, the FPGA is statically partitioned with each region being allocated a Circal process. Such processes define behaviourally self-contained components, while the assembly of processes defines the overall system's behaviour.

Modelling Circal circuits. The specified Circal processes are internally modelled as state transition graphs. This representation allows the VHM to easily determine the possible evolution of the process states. For example, the process P considered in the example is internally represented as the state transition graph depicted in Figure 5.

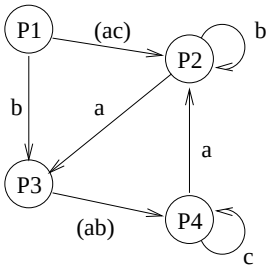


Figure 5. State transition graph for P

The nodes in this graph are in one-to-one correspondence with the Circal process definitions with arcs representing the guarded transitions to new states. Each node represents a state of the process and is thus represented using a so-called state block. The graph can be represented as a linked network of such state blocks. A particular state block thus leads to all possible next states by following the links that model the transition arcs in a breadth-first manner.

The data structure for a state block contains a pointer to each of the possible next states. Associated with each pointer is the guard for that transition. We also store the number of possible next states reachable from the present state as a means of quickly estimating the area needed to implement a circuit component. The implementation of circuit components and the estimation of their area is discussed in more detail below.

Process partitioning. Processes are partitioned according to their definitions. This is the most natural way to split a process given that a definition describes the behaviour of the process in a particular state. "Behaviour" here means how the process is able to evolve from the current state to a new state.

The state transition graph is thus logically partitionable into individual states and the set of arcs leaving a state. The state block is therefore also used to represent the unit of partitioning.

While the interpreter allows the logic for several connected states of a process to be implemented at once, it does not permit a state to be partially implemented i.e. if there is insufficient resource remaining to implement the logic for an additional state in its entirety, it is not implemented at all.

The interpreter is able to cope with the extreme case when resource constraints allow just one state for a process to be implemented. Similarly, the system allows the entire state transition graph for a process to be implemented when there is sufficient resource.

As an example, suppose the array area available to implement process P suffices to implement the behaviour of state $P1$, but no more. The circuit for $P1$ is illustrated in Figure 6. Note just those minterms and guard blocks that are needed to complete the possible state transitions from $P1$ are implemented. In order to determine which transition may have occurred, registers for states $P2$ and $P3$ are implemented as well, but it is not possible for the circuit to change state once it has entered one of these so-called *boundary states*.

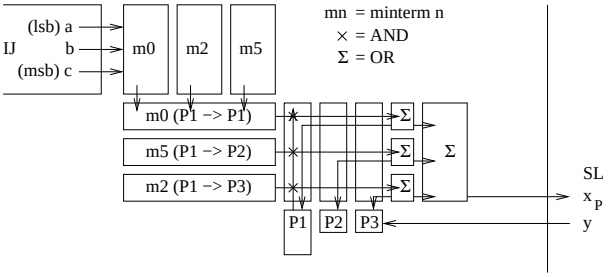


Figure 6. Module implementation of state $P1$

Determining what circuitry to load next. At initialization, the VHM divides the FPGA area into fixed sized regions and allocates one to each process. Each region is large enough to accommodate the logic for any single state block of the process allocated to it. If space permits, the region can accommodate the logic for several connected states (if not all of the states) for the process. The partitioning of the FPGA is discussed below in more detail.

As a computation proceeds, we would like to have that part of the state transition graph which the process is about to enter implemented on chip.

Due to resource constraints, there are situations when only a sub-graph of the complete state transition graph can be implemented. When the boundary of the sub-graph is reached, the interpreter retrieves a new sub-graph rooted at the boundary state that has become active.

When the locus of control for a process reaches a boundary state, i.e., a state that is at the head of an arc in the state transition graph, but which has not been implemented due to space constraints, the interpreter accesses that boundary state's description. By following the transition links it also retrieves the details of all possible next states, and so on,

in breadth-first order. The retrieval of state blocks and the construction of the corresponding circuit continues until the interpreter first fails to add an additional state due to space constraints.

In order to quickly determine which states can be implemented, the interpreter makes an estimate of the required space for a state based on the size of the sort of the process and the total number of terms in the process definitions of the states. This avoids back-tracking in circuit construction should it be discovered that an additional state cannot be accommodated.

The function used to estimate the area needed to implement the circuit for a sub-graph depends upon the mapping of the circuit to the target architecture. In calculating an estimate, let n_{imp} be the number of connected states of the process P that are being considered for implementation. Furthermore, let the size of the process sort be s, and for each of the i process definitions P_i contributing to the process, let there be t_i terms in the definition. Then for the XC6200 circuit implementations, the estimated width of the circuit implementing the sub-graph is at most

$$(T+1)\log s + T + 1 + \log n_{\text{imp}} + \log T, \tag{6}$$

where $T = \sum t_i$ for the states of the sub-graph to be implemented. The height of the corresponding circuit is at most

$$s + T + n_{\text{imp}} + 2. \tag{7}$$

The worst case impact of adding another state to the sub-graph being built is easily determined by including the number of terms, t_i, for the state in T and recalculating the above equations. The consideration of further states is abandoned as soon as the logic for a state cannot be added to the circuit under construction.

As the states to be implemented next are identified, the parameters defining its associated minterms, guard blocks, requesters, and state blocks are added to a list of sub-graph modules if they are not already included. For example, minterms and next states are often common to several definitions within the one process. When the sub-graph cannot be expanded, the module parameters are then used to generate fragments of the configuration for the corresponding circuit.

In the case of our example, let us assume the array area allocated to P suffices to implement $T = 6$ total sub-graph terms. The initial configuration can therefore accommodate the logic for states $P1$, $P2$, and $P3$, with $T = 5$, but the inclusion of $P4$, with $t_4 = 2$, is estimated to exceed the available space. The resulting circuit is depicted in Figure 7. In this case note that when the boundary state $P4$ is encountered, a configuration for the sub-graph rooted at $P4$, which includes states $P4$, $P2$, and $P3$, with $T = 5$, is created, and since this sub-graph is strongly connected, no subsequent reconfiguration is needed for process P.

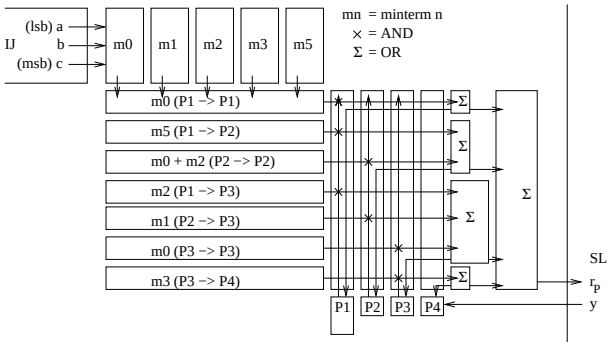


Figure 7. Module implementation of states $P1$, $P2$, and $P3$

Detecting the need for configuration swapping. Note that the circuit implementing the sub-graph includes a set of flip-flops that store the current state of the circuit (and, by implication, the process) using a one-hot encoding. A subset of these represents states that are included in the sub-graph, while the remainder represent boundary states that could not be accommodated in the breadth-first traversal of the state transition graph. When one of the boundary states becomes active, the interpreter forms a new sub-graph rooted at the active boundary state.

We currently envisage operating the system in one of two modes. Either the system is operated in an "observed" mode, in which the observer monitors the state evolution of the system, or the system is "unobserved", in which case its state is not monitored.

In the observed mode of operation the current state of the system is polled after each clock pulse. This involves reading all process state flip-flops to determine which state is active. In this mode of operation, the interpreter need only check whether the currently active state is implemented to determine whether a process needs to deploy a new region of the state transition graph.

On the other hand, if the system is operating in the unobserved mode, as part of an embedded system say, then it is feasible to associate with the boundary states additional logic that will interrupt the VHM when a new sub-graph is required. With static chip partitioning, it suffices to inform the VHM which region(s) issued the request. The VHM is then able to inspect the state flip-flops for the process allocated to each interrupting region in order to determine which boundary state has become active.

Ensuring configuration, design, and execution integrity. The pseudocode for the interpreter is listed in Figure 8.

```
/* Initialisation: */
analyze specification
partition FPGA area
for each process in turn
  define state transition graph
  set current state = boundary state = initial state
/* Operation: */
while system not halted
  /* configuration phase */
  for each process in turn
    if boundary state active
      calculate configuration rooted at boundary
          state and load
  /* execution phase */
  get event from environment and present to system
  allow system to respond to event in parallel
```

Figure 8. Interpreter operation

Execution integrity is guaranteed by ensuring circuit configuration and execution occur in separate phases of the operating cycle. By ensuring the logic for the current state, at least, is implemented on chip, the integrity of the design is maintained. We ensure the electrical integrity of the circuit that is not reconfigured by disconnecting a reconfiguring process circuit from the rest of the system at its Input Junction and Synchronization Logic. The circuit is not reconnected until the root state of the newly configured subgraph circuit is made active.

FPGA partitioning. The FPGA area available to the interpreter is statically partitioned. This approach reduces configuration overheads at run time since the region available to each process is guaranteed not to change.

The interpreter initially allocates sufficient space to each process for it to be able to implement its largest state block. Thereafter the region allocated to the process is expanded in order to provide more space for implementing additional states when possible.

The circuit area needs for a state block depend upon the implementation. From Equations 6 and 7 the width and height of the largest state block for a process can be determined if T is replaced by $\tau = \max\{t_i\}$ over all states for the process, and $n_{\text{imp}} = 1$ is substituted into the equations. For the XC6200 implementation, these substitutions allow the width of a state block to be approximated by $w = O(\tau \log s)$, and its height to be approximated by $h = O(s + \tau)$. From these approximations it can be observed that as τ increases the width of the circuit increases by a factor of $\log s$ more than its height. In order to implement more terms, as becomes necessary when additional states are to be added to a configuration, it is therefore primarily important to provide additional width.

Whereas the compiler stacks the processes above one another into a single strip, the interpreter packs the processes into multiple strips in order of decreasing width. Given an ordering on a set of allocation requests, a strip-packing stacks the requests in the given order into vertical strips beginning at the left edge of the available array. Within a strip, allocations are made starting at the top of the strip and extending to the base of the array. Allocations are alligned on their left boundary. If the FPGA height does not suffice to pack a request at the bottom of the current strip, a new strip is formed adjacent to the previous strip on its right. See Figure 9 for an example. If the processes cannot be packed into the available area, the interpreter stops. Otherwise there will generally be some free columns to the right of the packing and at the bottom of some of the strips.

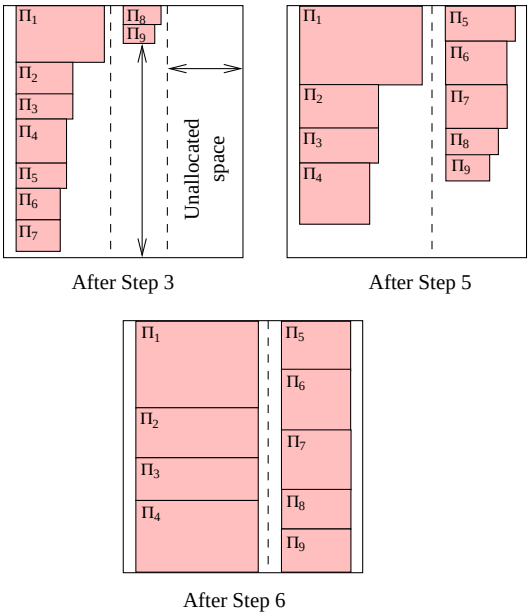


Figure 9. An example FPGA packing

Each allocation is subsequently expanded to fully use the available area. This expansion is carried out to allow the logic for additional states to be deployed. As described in the following algorithm, we expand the width of each block in proportion to the fraction of the FPGA's width left free. We expand the height according to the change in aspect ratio needed to accommodate additional logic as suggested by the approximations above.

Let (s_i, τ_i) be the sort size and the maximum over all states of the number of terms in a process definition for process Π_i. Let (w_i, h_i) be the width and height required to implement the largest state logic block for process Π_i as described above. Let the system to be implemented be composed of the n processes $\Pi_1, \Pi_2, ..., \Pi_n$. Define the width of the FPGA area to be W and the number of columns left

free by a strip-packing to be F. The algorithm for partitioning the available FPGA area is listed in Figure 10. Note the run-time complexity of the algorithm is at most $O(n \log n + n \log F)$.

```
/* initial packing */
1. given (s_i, t_i) for each Π_i, calculate (w_i, h_i)
2. sort the (w_i, h_i) list into decreasing width
3. strip-pack the FPGA and determine F
4. if F < 0
     abort the execution
/* expand requests */
5. if F > 0
     scale each (w_i, h_i) by e_w = W/(W − F) and
       e_hi = e_w / log s_i
     strip-pack the scaled list into the FPGA and
       determine F'
     if F' < 0
       halve F and repeat step 5.
6. increase the width of each process allocation to the
     width of its strip, and divide the free rows in each
     strip amongst its allocations
```

Figure 10. Partitioning algorithm

4. Conclusions

The interpreter is able to overcome FPGA resource limitations by partitioning and scheduling large circuits that implement abstract behaviours specified in Circal.

Implementation is being done now on a Virtex-targeted version of the compiler using an RCP1000 coprocessing board and JBits for elaborating and partially configuring the FPGA.

The VHM allows the interpreter to adapt to the available FPGA space. It works when just one state per process can be implemented. It can also implement the entire state-transition graph when space permits. In principle, the interpreter can also tolerate dynamic resizing of the process regions without additional overhead.

With the XC6200 module generators, the time needed to generate bit streams is proportional to the area of the region being reconfigured.

Strategies for potentially minimizing reconfiguration bitstream sizes would be useful. We have concentrated efforts on rapid circuit generation. These circuits overwrite the on-chip logic for a process i.e., do not reuse the on-chip logic. The penalty for doing so is additional configuration overheads.

One option for reducing run-time bitstream generation is to consider the possibilities for storing generated partitions so as to avoid regenerating them if they are needed again.

Dynamic chip partitioning strategies would potentially allow those processes that need more space to gain it, and are thus attractive. However, there is considerable overhead in managing a dynamic resource.

The application of these techniques to data-oriented applications is of considerable importance and will be investigated further.

We hope some of these techniques will apply to the implementation of dynamic process behaviours and structures. This is also to be further investigated.

References

[1] 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.

[2] O. Diessel and G. Milne. Behavioural language compilation with virtual hardware management. In R. W. Hartenstein and H. Grünbacher, 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*, pages 707 – 717, Berlin, Germany, 2000. Springer–Verlag.

[3] 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.

[4] J. G. Eldredge and B. L. Hutchings. Density enhancement of a neural network using FPGAs and run–time reconfiguration. In D. A. Buell and K. L. Pocek, editors, *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'94)*, pages 180 – 188, Los Alamitos, CA, Apr. 1994. IEEE Computer Society Press.

[5] P. Lysaght, G. McGregor, and J. Stockwood. Configuration controller synthesis for dynamically reconfigurable systems. In *IEE Colloquium on Hardware–Software Cosynthesis for Reconfigurable Systems*, pages 1 – 9, London, UK, Feb. 1996. IEE.

[6] D. Robinson and P. Lysaght. Modelling and synthesis of configuration controllers for dynamically reconfigurable logic systems using the DCS CAD framework. In P. Lysaght, J. Irvine, and R. W. Hartenstein, editors, *Field–Programmable Logic and Applications, 9th International Workshop, FPL'99 Proceedings*, volume 1673 of *Lecture Notes in Computer Science*, pages 41 – 50, Berlin, Germany, 1999. Springer–Verlag.

[7] M. J. Wirthlin and B. L. Hutchings. Sequencing run–time reconfigured hardware with software. In *FPGA'96 1996 ACM Fourth International Symposium on Field Programmable Gate Arrays*, pages 122 – 128, New York, NY, Feb. 1996. ACM Press.