Very High-Level Synthesis of Datapath and Control Structures for Reconfigurable Logic Devices*

Extended Abstract

Pablo Moisset, Joonseok Park, Pedro Diniz

USC / Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, CA, 90292
{pmoisset,joonseok,pedro}@isi.edu

1. Motivation

Reconfigurable logic devices offer the potential of performance improvement through their ability to be configured for specialized operations. Examples include the definition of custom datapaths to handle non-standard arithmetic computation or the implementation of specific fine grain bit-level preprocessing that would otherwise require very large amounts of input bandwidth.

The widespread adoption and realization of the potential offered by reconfigurable logic devices (e.g, delivered by FPGAs) in the realm of embedded computing requires the existence of adequate tools that can take applications written in high-level programming languages such as C or Matlab, are directly translate them into an HDL. Current commercially available tools [1] are able to translate program in C and generate a behavioral VHDL equivalent but generate very generic (and therefore low quality) control structures. As a result of the poor performance engineers have to code their control hardware to meet the datapath design defeating the whole purpose of isolating the programmer from the low level details of hardware specification.

In this paper, we describe our approach to the synthesis of inner loops that manipulate array variables using affine index access functions written in C directly to VHDL. Our approach uses high-level data dependence analysis techniques that allow our compiler to apply a variety of transformations for space or execution time reduction. Our approach is fully automatic and generates both the datapath that implements the functionality of the loop body as well as the structures that control the execution of the datapath using either pipelined or non-pipelined execution. We believe this separation of concerns between the datapath and the control will allows us to generate several control strategies while keeping a separation of concerns therefore promoting overall design reuse.

We structure this paper as follow. In the next section, we characterized the loops our analysis is able to handle. Next, we describe the basic compiler analysis and transformations using a simple example. The next section describes the strategy used to generate control structures and all of the supporting parameterized control blocks used. We report on the status of our system implementation and then conclude.

2. Synthesis Example

We illustrate the application of our analysis and synthesis approach to a moving average computation pervasive in digital signal processing. We describe to apply compiler analysis and transformations to automatically synthesize an efficient datapath.

```
for(i=0; i < N; i++)
   b[i] =(a[i]+a[i+1]+a[i+2])/4;
```

Given the C code above the compiler can trivially generate a datapath for the body corresponding to the summation of the values of three registers as depicted in the figure 1. This translation closely resembles an abstract syntax tree, replacing variable accesses with registers.

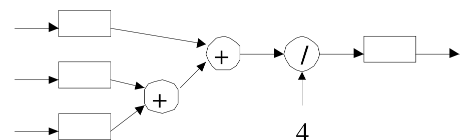


Figure 1 Straightforward Implementation.

* This research was sponsored by DARPA under contract F30602-98-2-0113.

The control implementation for this example would have to fetch three input values for each output value produced.

While simple to generate both the datapath and the corresponding control from a for loop specification, this approach wastes significant I/O bandwidth. For every value output the control must feed three data into the circuit. However, for a given iteration of the loop, the compiler may realize that two out of the three values required for the subsequent iteration are already in registers, although in a different place. The solution is to stream the data using a FIFO as show in Figure 2. For each output value produced only a single new input value is needed.

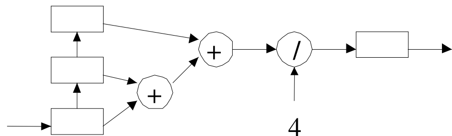


Figure 2. Using a Queue to Reuse Input Data.

To accomplish this transformation the compiler uses data dependence analysis to capture the overlap of several array references in the loop [Wolfe95].

A compiler can further improve the performance of the datapath in Figure 2 by applying pipelined execution techniques. The newly inserted registers reduce the critical path of the datapath and hence increase the attainable clock rate with the corresponding increase in throughput.

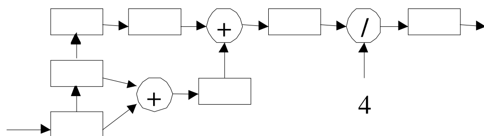


Figure 3. Pipelined Version of Average Filter.

The problem of determining where to insert the registers can be mapped into an Integer Linear Programming problem. The magnitudes considered are area and clock rate. One magnitude is used as objective function, and the other as a constraint as described in [Wein97] and [Shen96].

The datapath in Figure 3 can be further improved. For this particular example the compiler can recognize commutative and associative operator and abstract n-ary operators like the one in Figure 4.

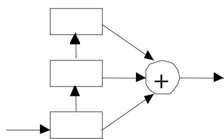


Figure 4. Three Input Adder from Queue.

The original C code contains two binary addition operations as part of the loop body implementation. The computation, however, can be viewed as the addition of three consecutive elements of a sequence. Given more abstract description of the computation, allows the generation of a more efficient. In our example, the adders, the input queue and the registers used for pipelining can be replaced with the following design.

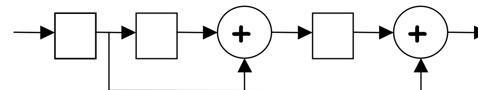


Figure 5. Alternative Implementation.

This implementation saves two registers over the implementation in Figure 3. This transformation can be used with any commutative and associative operator as well as with several bit-wise operations.

Although these transformations are well known to our knowledge there is no available commercial synthesis tool capable of automatically derive a high-quality datapath and control from a for loop specification without constants loop bounds. Existing commercial tools will either unroll the loop if the bounds are known (and subject to space constraints on the target hardware) or will at most generate VHDL specification of the loop embedding both the datapath functionality and the control. Although the resulting code is equivalent to the original code specification in C (in terms of simulation) the resulting synthesized hardware (when the tool supports it) is extremely poor in terms of compactness of the design.

3. Datapath Generation

Our analysis for the generation of datapath focuses on loops with constrained bounds of incremental functions. In the current implementation simplicity we have constrained the generation of the datapaths for loops with a single basic block in the body of the loop. The abstract syntax tree is converted into a DAG, by replacing variable accesses, in the right hand side of assignments with a link to the last definition of that variable, if any. After all replacements are done, only locally exposed definitions are processed.

For each array read in the body, the compiler uses *input data dependence* analysis and creates one or more queues. The number, and length of them is computed easily from the indices. At this point, the compiler may make some decisions involving the trade-off between I/O bandwidth usage and the number of registers used in queues. It is possible to eliminate long queues or to reduce the length of one queue at the cost of more I/O operations. Scalars can be considered arrays of one element.

3.1. Loop Carried Dependences

If one value generated during the execution of an

iteration is used in a following one (iteration) there is, in compiler terminology, a loop carried dependence. To avoid taking generated data out of the datapath and inserting it again at the beginning of a subsequent iteration, a cyclic datapath is convenient. Our algorithm will generate a correct datapath under this circumstance. Once the input queues are created, it is easy to detect a loop carried dependency. If a datapath output matches an input, then a loop is created, otherwise an output register is created. Inserting registers in cycles to improve the clock rate is impossible, but sometimes it is possible to pipeline the acyclic regions of the datapath.

3.2. Datapath Synthesis Algorithm

Our datapath synthesis algorithm is currently limited to inner loop that uses array with affine index where all array accesses to a given array structure are of the form $a[\alpha_a * i + \beta_k]$ and where α_a and β_k are constants.

For every array structure in the computation, our analysis will generate one or more input queues. The number of queues and their length is given by the equation below where Card(S) means the cardinality of the set S.

$$N_a = \text{Card} \ \{ \ x \ / \ x = \text{Remainder}(\ k \ / \ \alpha_a) \text{ for all valid k's } \}$$

If there are two accesses $a[\alpha_a * i + \beta_{k1}]$ and $a[\alpha_a * i + \beta_{k2}]$ such that Remainder(k1 / α_a) = Remainder(k2 / α_a), then both array elements will be taken from the same queue. The length of each queue is computed as:

$$L = (\max(\beta_k) - \min(\beta_k)) / \alpha_a + 1$$

considering β_k's of array accesses from that particular queue only.

If there are loop carried dependences we extend the queues with multiplexors in order to allow the insertion of newly computed elements into the middle of a queue. Figure 6 below illustrates the synthesis of a queue for a computation with loop carried dependences (this extension is not yet implemented).

```
for(i=0; i < N; i++)
     a[i] =(a[i-1]+a[i]+a[i+1])/4;
```

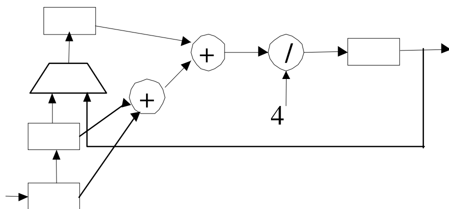


Figure 6. Queue Synthesis with Loop-Carried Dependences.

4. Control Generation

We now illustrate how to derive the control signal for a given datapath. Our analysis uses the value of the latency of the individual functional nodes (e.g., adder or multiplier) in a datapath to derive a suitable timing schedule. If pipelining execution is required in our analysis, we construct a schedule for prolog, steady state and epilog as software pipelining [Lam88].

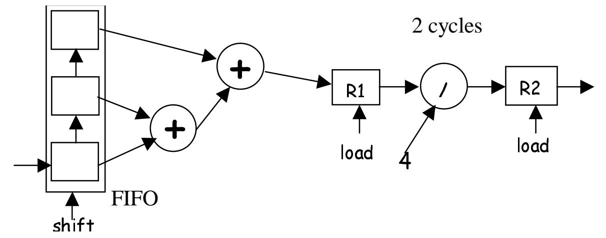
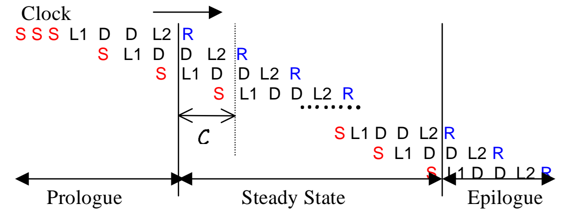


Figure 7. Datapath with Control Signals

For this particular example, the proper control signal streams are shown in Figure 8.



S: shift & input data, ***Lx***: load register x, ***D***: shift right 1 bit, ***R***: store result to local memory, ***C***: steady state cycles

Figure 8. Control Signals for Pipelined Execution

Repetitive actions in steady state depend on the greatest latency block in datapath. To make control signals for those blocks, we need predefined control signal packages and clock latency of that block in our library –in this example, "divide by 4" requires two cycles with "shift right bit" control signals. Loop bounds and step value can determine the repetition of steady state cycles. We can schedule prolog and epilog according to the shape of datapath and the FIFO queue size for input data. In the general case, we can extract the stage cycle information with the following equations.

Prologue: $d + Q$

Steady state iterations: $N - \left\lceil \frac{d}{c} \right\rceil$

Epilogue: $\left(\left\lceil \frac{d}{c} \right\rceil - 1 \right) * c + 1$

Total steady state cycles : $c * \left(N - \left\lceil \frac{d}{c} \right\rceil \right)$

d : clock latency of a single during steady state.
Q : extra clock cycles to fill input FIFO.
c : steady state period.
N : number of iterations.

Using this schedule, we derive the FSM that issues the control signals of the datapath.

There are several approaches to generate control signal. Two popular and simple approaches are (1) a state-table-based method where the VHDL specification enumerates all the states and transition for the prologue and epilogue, (2) a counter-based approach as described in where part of the state enumeration is done via counters [Hayes88].

The counter-based approach more modular than the first approach. As we are targeting FPGA-based hardware platforms the counter-based approach has the additional advantage of allowing for the reuse of the counter across multiple datapath controllers. The first choice very unlikely would offer such possibility.

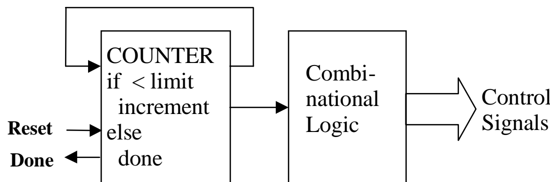


Figure 9. Counter Based FSM.

Table 1 illustrates the differences in the size and complexity of the two control generation strategies. Table 1. illustrates the space and time performance of these two control generation strategies. As the table reveals the counter-based method uses much fewer number of FFs and a substantial lower number of gates than the state-table-based method (using one-hot encoding). Although the performances are comparable (87MHz vs. 116 MHz) we were not able to attain a complete (100%) timing analysis path coverage for the table-based method.

Method	Counter-based	State-table-based
LUTs	44	45
FFs	13	27
CLB	22	23
Gates	368	465
Freq(MHz)	87	116

Table 1. Size and Complexity of Controller for Moving Average Computation Example (prologue, steady state and epilogue).

5. VHDL Code Emission

In the last step of our framework, we have VHDL code generation module. For each datapath, it generates one structural (hierarchical) VHDL code in the highest level. While traversing data flow inside the datapath, we instantiate functional element and translate control information into signal (wire) and component instantiation statements. For each functional element in the datapath, we translate it into predefined behavioral VHDL code, which is fully parameterized.

In addition, we can generate control FSM in VHDL code. We need four counters - two for steady states and one for prolog & epilog - and control signals. Our control generator is Moore type FSM. Each stage is represented by counter number and combinational logic generates control signals.

6. Status

Only loop bodies are considered for synthesis, there can be array accesses, but the indices are limited to affine functions of the index variables. The control structure must be simple, that is, we only allow assignments and if statements to be in the body.

Our current version can deal only with single loop bodies without if statements. Input data reuse is implemented, as well as the VHDL code generation. The loop body will be allowed to include conditional statements, the datapath will represent them using multiplexors that will select the correct side effect based on the evaluation of the condition. Although single loops accessing one-dimensional arrays may suffice in many cases, sometimes support for more complex cases is required. An example is:

```
for (i = ...
   for (j = ...
      for (k = 0; k < 2*i+3*j; k++)
         a[3*i+j, 2*k, 1+j] = ...
```

We plan to extend the compiler to synthesize datapath for perfectly nested loops, accessing multi-dimensional arrays using affine expressions as indices. Data reuse is more complicated because the model of input queues is not general enough to handle the general case. There are more opportunities for optimization, because techniques as skewing and loop permutation can be used to improve data locality. The theory is described in [Wolf91]. The SUIF compiler already includes most of the analysis routines needed here.

Our VHDL generation module can emit limited kinds of functional element and our control generator currently targets a simple pipeline control. In the future we plan to extend the type of operators, and increase the sophistication of the pipeline control, in particular considering the relationship between memory addressing and access timing control and the control of the pipelining.

7. Related work

Researchers at Carnegie Mellon University developed the PipeWrench [Shmidt97] reconfigurable architecture. To implement application in this architecture they use the DIL programming language. The DIL language is a single assignment language. It includes the abstraction of delayed assignment. This abstraction allows the programmer to pipeline the datapath manually. Control is constrained by mapping several concurrent operators to each of the virtual reconfigurable strips of the pipeline. The routing and possible mapping of the data input and data output ports are defined by the configuration of each stripe. Pipeline control is programmable and embedded in the PipeWrench architecture.

The Garp project at the University of California, Berkeley, also targets automatic compilation of C programs to the Garp reconfigurable system. The Garp compiler is based on SUIF and targets exclusively the Garp chip which includes a specific place and routing implementation for the reconfigurable fabric of the Garp chip.

The research effort described in this paper relies on commercial tools to create configurations for commercially available FPGA hardware.

A number of commercial synthesis tools for FPGAs claim to handle loop constructs. We have found however that they generate VHDL geared toward simulation rather than synthesis. As a result the generated VHDL code is a mix of both the datapath functional specification and the control specification. This makes the generated VHDL hard to understand limiting its programmer portability.

The approach taken in our research aims at separating, to a given extent possible, the specification of the functional datapath from the control specification. This separation of concerns allows us to generate several control strategies while keeping a separation of concerns that promote design reuse.

8. Conclusions

This paper presented an overview of the required analysis techniques and transformations that allow a tool to generate both datapath and controllers for loops that manipulate array variables using affine index access functions. We believe the integration of existing analysis and the development of new compiler analysis techniques and compilation strategies will greatly facilitate the migration of applications to reconfigurable computing platforms.

References

[Lam88] M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," In Proc. of the SIGPLAN'88 Conference on Programming Languages Design and Implementation, June 1988.

[Wolf91] M Wolf and M. Lam, "A loop transformation theory and an algorithm to maximize parallelism". IEEE Transactions on Parallel and Distributed Systems, 2(4), pp. 452--470, October 1991.

[Schmidt97] Herman Schmidt, "Incremental Reconfiguration for Pipelined Applications", In Proceedings of the IEEE Symp. On FPGA for Custom Computing Machines, Napa, CA, 1997.

[Shenoy96] Shenoy, "Retiming: Theory and Practice", In the Integration VLSI journal, 22, pp. 1-21, 1997.

[Wein97] Markus Weinhardt , "Pipeline Synthesis and Optimization for Reconfigurable Custom Computing Machines", Universitaet Karlsruhe, Fakultaet fuer Informatik D-76128 Karlsruhe, Germany January 3, 1997.

[Hayes 88] J. Hayes, Computer Architecture and Organization, McGraw-Hill, 1988.

[Wolfe95] M. Wolfe, High Performance Compilers for Parallel Computing, Addison-Wesley, 1995.