The synthesis of a hardware scheduler for Non-Manifest Loops

Omar Mansour   Egbert Molenkamp   Thijs Krol

University of Twente, Department of Computer Science

P.O. Box 217, 7500 AE Enschede, the Netherlands

Phone: +31 (0)53 4894178   Fax: +31 (0)53 4894590

E-mail: {mansour,molenkam,krol}@cs.utwente.nl

Abstract

This paper[1] addresses the hardware implementation of a dynamic scheduler for non-manifest data dependent periodic loops. Static scheduling techniques which are known to give near optimal scheduling-solutions for manifest loops, fail at scheduling non-manifest loops, since they lack the run time information needed which makes a static schedule feasible. In this paper a dynamic scheduling approach was chosen to circumvent this problem. We present a case study using VHDL were the focus lies on implementations with minimal memory usage and low communication overhead between various components of the architecture. This has resulted in an efficient and synthesisable system.

Keywords: *non-manifest loop scheduling, dynamic hardware scheduling.*

1 Introduction

High-level synthesis translates behavioral descriptions written in a high level language (*HLL*) such as C or C++ into hardware structures described by VHDL, or Verilog. This translation starts by converting the behavioral description to a control data flow graph *CDFG* [4] and then performing a number of optimization's such as *dead code removal, constant propagation, common sub-expression elimination, tree height reduction, code motion, loop unrolling, in-lining* and finally *scheduling* and *allocation* onto hardware resources. In digital signal processing (*DSP*) and video signal processing (*VSP*) applications, many algorithms have a repetitive and periodic nature [2] the same computations must be executed on arrival of each new data sample or block of samples. Some loops within a computation require a constant

number of clock iterations in their loop-body and their number of iterations is fixed, they thus have a fixed total execution length. Such loops are called manifest-loops. Non-manifest data dependent loops, on the other hand, are those where the number of iterations required in order to perform a computation is data dependent and hence have a variable execution length which is not known at compile time. The Euclidian **gcd(x,y)** algorithm shown in Fig.1 is a typical example of such loops. For 16 bit input values the algorithm requires a variable number of iterations ranging from 1 upto 23 cycles. Where the actual number of iterations is a function of the input values x and y.

```
int gcd(int x, int y){
    int g;

    assert ((x>0) && (y>0));
    g = y;
    while ( x > 0 ){
        g = x;
        x = y % x;
        y = g;
    }
    return (g);
}
```

Figure 1. gcd algorithm

Functional units can be classified into two classes. Functional units which are (1) *analytic* and (2) *non-analytic* or *soft* functions. Analytic functions are those that have one exact answer and in order to calculate that answer a variable number of clock cycles, which is dependent of the input data, is needed. Non-analytic functional units, on the other hand, converge to the required result in time. The quality of the result is improved upon in each iteration. Examples of such units are the Taylor expansion series, MPEG decoding, and the CORDIC-rotation algorithm. When scheduling loops of non-analytic functions one can statically schedule the loop and set its execution to a fixed number of iterations. The quality of the result in the case of too few iterations would be sacrificed and in the case of too many iterations we lose the remaining valuable clock cycles. This shows

us that static scheduling of non-manifest non-analytic functional operations is not without cost. Formal techniques for solving the scheduling problem of manifest loops using Integer Linear Programming (*ILP*) have been devised in [2]. By modeling the periodicity of operations using a bounded period vector denoting the periods between two consecutive iterations, determining a start time and processing unit on which the operation is to be executed, a feasible scheduling solution under area-, processing unit-, and timing- constraints could be found. Since the *ILP* technique requires prior knowledge of the operation's duration in order to perform, this technique is infeasible for the scheduling algorithms with non-manifest loop behavior. We must mention that the loop algorithms operate on data input streams (block's) with no data dependencies between various loop-computations. In other words each input value results in a separate, non-manifest, computation independent of previous computations. In [1] the underlying theory of the formal scheduling problem was constructed. In section 2 we mention some related work on hardware schedulers and variable latency components. A summary of the derived definitions and formula s is given in section 3. In section 4 we describe the basic hardware architecture of the scheduler. In section 5 we provide a case study were memory usage of the scheduler and the communication overhead between the resources of the scheduler and the controller have been optimized. Section 6 provides some experimental results and finally the conclusions and future work are provided in sections 7, and 8.

2 Related Work

Variable latency components are in a way similar to non-manifest loops. Both types of components have a data-dependent latency. In the case of variable latency components, the latency of the loop-body is data dependent and the number of iterations is a constant. In the case of non-manifest loops, the latency of the loop-body is a constant and the number of loop iterations is variable based on the input data.

Silvia M. Mueller [3] addresses the problem of dynamically scheduling the body-part of a variable latency functional-unit. In her work she mentions that the scheduling of multiple functional units can be split up into two parts: (a) Global scheduling, this scheduling governs the interaction between the functional units. Many scheduling algorithms can not cope with variable latency units as they require prior knowledge of the latency and the required resources. However schedulers based on the Tomasulo algorithm make no prior assumption of the functional units latency and hence are suited for the global scheduling problem. (b) Local scheduling, which is the scheduling of resources within a variable latency unit. Due to multiple paths within the body-part of a variable latency unit, a situation

can exist were instructions would compete for resources, it is the task of the local scheduler to ensure that within a functional unit there are no contentions on the busses, and that no data is lost. The local scheduler devised, schedules instructions competing for a resource based on their age. The oldest instruction in the stream gets the resource. This ensures that the latency of the functional units is never increased as in the case of simple FIFO queues based schedulers.

Vijay Raghunathan, Srivaths Ravi, and Ganesh Lakshminarayana [6] address the problem of integrating variable latency components into high-level synthesis. In their work they show that with variable latency components throughput could be gained if the components were properly placed on the critical paths. Improper placement of the components could lead to decreased throughput. Since extra overhead imposed by variable latency components usually leads to increased chip area, they present a technique to further reduce the chip area overhead. This technique is based on the concept of reduced latency units.

L. Benini, E.Macii, M.Poncino, and G.De Micheli [7] address the performance issues related to the optimization of VLSI designs. In their work they introduce the concept of *telescopic units*, which is in essence another term for variable latency units. They show that every constant latency unit can be transformed into a variable latency unit by adding a signal for detecting completion. In this case the output of the circuit is available as soon as it is ready as opposed to being constrained by the worst case delay of the circuit.

3 Background Theory

In a real time embedded system the latency of the individual computational units play a big role and it is important to know the latency of the system in advance. If the computational units of the system have a non-manifest behavior, it is difficult to statically determine the latency. The approach presented in this paper is to determine the required latency and the number of computational resources based on prior knowledge of the work load needed in order to process the input data. In [1] the following formal problem description was presented: Given an input data stream with known maximum workload bound B on a stream window of size m, hence $WL(t, m) \leq B$, and a data dependent non-manifest loop algorithm A with known bounds CL_{min_A} and CL_{max_A}, devise a real time hardware scheduler that will meet the workload $WL(t, m)$ of the system, produce an output that is synchronous with the input in a time frame of at most m time units, and finally determine the resources of the system and their allocation.

Further in the problem formulation it is assumed that each time unit is equivalent to a single computation cycle of the non-manifest loop and that each resource, is capa-

ble of performing one computation cycle within one time unit. $CL_A(v)$ denotes the number of computation cycles algorithm A would require in order to perform its computation on the input value v. CL_{max_A} is the maximum number of computation cycles needed by algorithm A. $WL(t, m)$ is the accumulated workload on an input stream-window of length m for a given algorithm A. B is the maximum, or upper bound, of the accumulated workload.

The following relations hold:

$$WL(t, m) = \sum_{j=t}^{t+m-1} CL_A(v_j) \qquad \forall t, m \in \mathbb{N} \quad (1)$$

$$WL(t, m) \leq B \quad (2)$$

And finally we state that:

$$m \geq CL_{max} + \left\lfloor \frac{B - CL_{max} + \frac{N_{res} \times (N_{res}-1)}{2}}{N_{res}} \right\rfloor - N_{res} \quad (3)$$

$$N_{res} \leq CL_{max} \quad (4)$$

Where N_{res} is the number of resources available to the scheduler. For proof of equation 3 see [1].

In order to devise a hardware scheduler for a given algorithm A and known CL_{max} value we iterate on various values of m and N_{res} until an optimum solution, in terms of either window size or minimum number of resources, is found which will satisfy equation 3.

4   Initial hardware model

This section presents the initial hardware model of the scheduler and its resources. We describe the model by means of an illustrative example.

Example: An algorithm A has to process input samples on each clock cycle. From the specification of the data input stream we know that the accumulated work load on a window of length 14 clock cycles is less than the upper bound B which is 30 clock cycles, hence $WL(t, 14) \leq 30$, and the specification of algorithm A specifies that the maximum number of iterations needed for a single computation CL_{max_A} is 10 clock cycles. In order to find the required number of resources which are capable of handling the specified workload we iterate on N_{res} in equation 3 we find that N_{res} must be at least 3 (See table 1).

Figure 2 presents a block model of the constructed hardware scheduler and its resources. The model consists mainly of the following parts:

- data queue

- time queue

- time counter register

Table 1. The scheduling constraints, input data stream specifications

Load type	range	units
$CL(v)$	1..10	CLKs
$WL(t, 14)$	≤ 30	CLKs
m	14	CLKs

- start time resource registers

- computation resources $\{R1 \ldots R3\}$

- address selection unit

- FIFO reorder buffer

- data routing multiplexers

Since execution of a computation has a variable length. Some computations will produce their output at an earlier stage than their predecessor computations. If this occurs, the produced output stream is out of order. It is the functionality of the address selection unit and the FIFO-reorder buffer to ensure synchronization in such a way that the output stream is in order with the input stream. This is accomplished by writing the result at different addresses within the reorder buffer based on the actual amount of clock cycles consumed by the resource during its computation.

The model behaves as follows, on each clock cycle the time register is incremented; the input data will be allocated to either the wait queue or to a free resource if available. If the input data is allocated to the wait queue the accompanying value of the $Time$ register is stored in the time queue, this ensures that we preserve the time of arrival, of the input data. Depending on the number of free resources, one, two or three data items will be allocated to the free resources. If the resources have produced their outputs the address selection will take place and either one, two, or three data values will be written to the reorder buffer at different addresses. The reorder buffer positions are shifted, one place, and the value of reorder buffer at position zero will become the output of the system, which basically forms the output stream. Address selection is accomplished by the following equation:

$$Ad = m - \mid T_{current} - T_{start} \mid \quad (5)$$

5   Generic scheduler case study

This basic hardware model has a number of short comings. First, if the incoming data cannot directly be allocated to a resource, because all resources are occupied with previous computations, it will be placed in an input queue together with the time of its arrival. And once a resource is
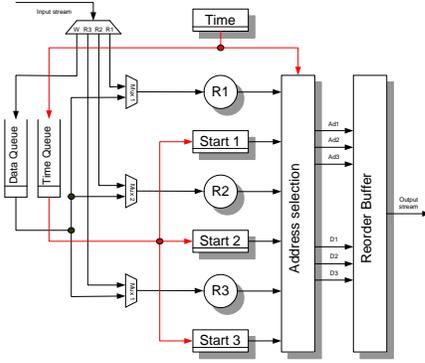
Figure 2. The hardware model of the scheduler, the controller and all control signals are omitted from the figure for simplicity reasons

free this input data will be allocated to the free resource for processing. The resource will be busy with the processing for at most CL_{max} cycles. The number of memory places within the reorder buffer is equivalent to the latency of the system which is m. This implies that, if an input data is in the wait queue there is always a free place for that data within the reorder buffer. The basic hardware model does not make use of this property and hence there is a memory overhead within the system. Secondly, if we devise a system were the number of memory elements is equal to the latency m, and by allowing the outputs and inputs to be read and then written to each memory element in a cyclic fashion, we can get rid of the input queue, time queue, and address selection unit of Fig.2.

In this section we examine an improved model. The scheduling system operates (see Fig.3) as follows: assume we have n memory elements organized as a cyclic queue and that $first$ and $last$ are indices within the range $[1 \ldots n]$. At the beginning of the system $first$ and $last$ will point to the same memory address. The scheduler performs the following tasks in a cyclic fashion.

First, the scheduler will read the value within memory element $first$ and produce that value on the output stream of the system.

Second, the scheduler will scan the resources to see whether they are done with their previous computations. If some resources have completed their computation, the scheduler will write their outputs into the memory array using the information stored in an allocation table.

Third, the scheduler will try to allocate the input data to a free resource. If there are no free resources, the newly received input data will be written to the memory element indexed by $first$ (in this case the memory behaves as an input queue) and the $first$ index will be updated such that it points to the next memory element of the array in a cyclic

fashion. If there are free resources, the scheduler will iterate on the waiting data, which is the data within the consecutive memory range of indices $last$ and $first$, assigning them to the free resources. Once data has been assigned to a free resource the scheduler will maintain this information in the allocation table.

Finally if there are still free resources and there is no more data waiting within the memory (indicated by $last$ and $first$), the scheduler will allocate the received input data to a free resource and maintain this information in the allocation table. The resources will then perform the required computation, within at most CL_{max} clock cycles. The resources will indicate whether they are active or not using an active signal, which is monitored by the scheduler. The scheduler uses the active signal, of the resources, and the information in the allocation table to decide whether a resource is available or not.
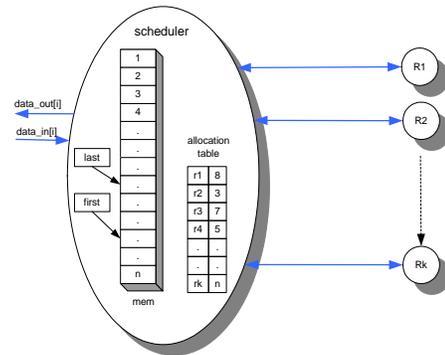

Figure 3. The new hardware model

Communication between the scheduler and the resources commence via a number of separate busses. The bus-signals are depicted in Fig.4.
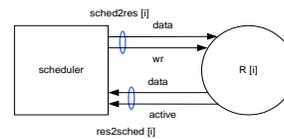

Figure 4. Communication busses between controller and resource

One of the problems we face is the communication overhead between the resources and the scheduler. If the communication is synchronous, it will take one clock cycle extra for each wr action. If we assume that each iteration of a non-manifest computation takes one clock cycle, than the complete computation will take at most $CL_{max} + 2$ clock cycles. This means that the scheduler will not meet the latency m shown in equation 3. One solution to this problem is to modify equation 3 accordingly:

$$m \geq$$
$$CL_{max} + 2 + \left\lfloor \frac{B - (CL_{max}+2) + \frac{N_{res} \times (N_{res}-1)}{2}}{N_{res}} \right\rfloor - N_{res}$$

This will allow us to maintain synchronicity at the cost of extra latency. Another solution is to design the resource as a Mealy model and the scheduler as a Moore model (see Fig.5).
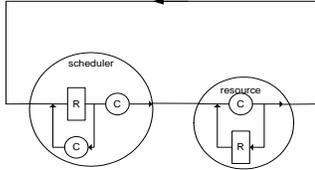

Figure 5. Communication model between scheduler and resource using Mealy and Moore models

In this model, communication within one clock cycle will have a path with only one register which is within the scheduler. This means that if a computation takes one iteration it will consume one clock cycle[2].

5.1 Implementation of the resource

In this subsection we describe how to implement the resource as a Mealy model. The specification of the resource is as follows: The resource will behave as a non-manifest loop and at the same time, the number of loop computations must be controlled by the input data. Hence, if the scheduler provides the data value v where $v \in [1 \ldots CL_{max}]$ the resource will consume the amount v iterations (clock cycles) during its computation. The result of the computation is always the same value v provided as its input.

This allows us to test the scheduler under various load conditions by providing an input stream with known computation load (the sum of the input values is the provided load of the system, see equation (1)). Table 2 shows the specification of the required resource.

In Fig.6 the unfolded version and in Fig.7 the folded version of a generic resource are shown. The design process starts by unfolding the specification in time, and in order to design the resource as a Mealy model, the path from input to output, for a single iteration computation, must be within the same time unit. In other words providing an input to the resource at time t_i where the resource would provide its output at time t_{i+1} is not allowed.

---
[2]**Note:** Although the communication overhead is lower than the synchronous communication solution, the critical path is probably longer.

Table 2. Specification of the resource

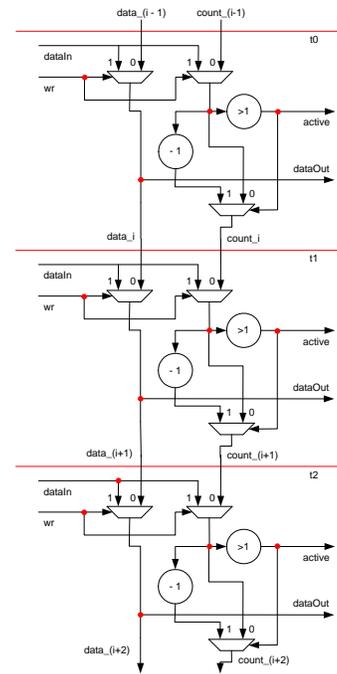| control	action
wr & -	$data_i = dataIn$ $count_i = dataIn$ $active = dataIn > 1$ $dataOut = dataIn$
$\overline{wr}\&active$	$data_i = data_{i-1}$ $count_i = count_{i-1} - 1$ $active = count_{i-1} > 1$ $dataOut = data_{i-1}$
$\overline{wr}\&\overline{active}$	$data_i = data_{i-1}$ $count_i = count_{i-1}$ $dataOut = data_{i-1}$

Figure 6. Generic resource unfolded in time

The second step is to fold the resource in time. Registers will be added to data signals which cross the time line. Finally we give names to all the internal signals and write the VHDL description of the resource based on the folded version of the resource shown in Fig.7.

In the VHDL implementation of the resource we use the data type *DataTp* this is declared in the package type shown in Fig.9. The package mainly contains the needed data structures and some functions used by the scheduler.

5.2 Implementation of the scheduler

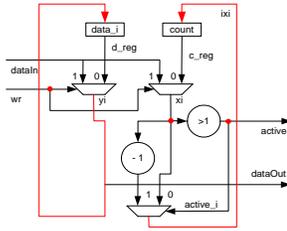The scheduler's VHDL code is shown in Fig.10. It consists of a memory array which is used to store the data, an

Figure 7. Folded version of the generic resource

```
ENTITY res IS
  PORT(clk       : in  std_logic;
       reset     : in  std_logic;
       wr        : in  std_logic;
       active    : out boolean;
       DataIn    : in  DataTp;
       DataOut   : out DataTp);
END res;


architecture behavior of res is
signal d_reg    : DataTp;
signal c_reg    : DataTp;
signal xi       : DataTp;
signal yi       : DataTp;
signal active_i : boolean;
signal ixi      : DataTp;
begin
  process (clk, reset)
  begin
    if rising_edge(clk) then
        c_reg <= ixi;
        d_reg <= yi;
    end if;
  end process;
  xi <= dataIn when wr='1' else c_reg;
  yi <= dataIn when wr='1' else d_reg;
  active_i <=  (xi > 1);
  dataOut <= yi;
  ixi <= xi - 1 when active_i = true else xi;
  active <= active_i;
end behavior;
```

Figure 8. VHDL code of a generic resource

allocation table, for keeping track the computations resultant memory address.

The controller basically performs the following tasks in a repetitive manner.

- write the result data on to the output stream

- store the valid data from the resources to their appropriate memory locations

- allocate waiting data to free resources

The scheduler configuration is given in Fig.11. It basically describes how the connections between the scheduler and the resources are established.

6 Experimental results

In this section we show scheduling-simulation results for the gcd resource shown in Fig.12. Using the simulation tool ModelSim from Model technology [5] the simulation

```
PACKAGE types IS
  CONSTANT Nres : positive := 3; -- number of resources
  CONSTANT M    : positive := 14; -- Latency

  --- Data dependent properties
  CONSTANT maxint : integer := 2**4-1;
  SUBTYPE int0tomax IS integer RANGE 0 TO maxint;
  TYPE DataTpIn IS ARRAY(0 TO 1) OF int0tomax;
  SUBTYPE DataTpOut IS int0tomax;

  -- Memory for storing data
  TYPE Storage IS RECORD
    inp  : DataTpIn;
    outp : DataTpOut;
  END RECORD;
  TYPE MemoryTp IS ARRAY(0 TO M-1) OF Storage;

  -- Type used for data transport between resources and scheduler
  TYPE DataVecTpIn  IS ARRAY (0 TO Nres-1) OF DataTpIn;
  TYPE DataVecTpOut IS ARRAY (0 TO Nres-1) OF DataTpOut;
  TYPE IndexVecTp   IS ARRAY (0 TO Nres-1)
   OF integer RANGE 0 TO M;
  TYPE DataSchedResTpIn IS RECORD
    Data  : DataTpIn;
    wr    : std_logic;
  END RECORD;
  TYPE DataSchedResVecTpIn IS ARRAY(0 TO Nres-1)
   OF DataSchedResTpIn;

  TYPE DataSchedResTpOut IS RECORD
    Data   : DataTpOut;
    active : boolean;
  END RECORD;

  TYPE DataSchedResVecTpOut IS ARRAY(0 TO Nres-1)
   OF DataSchedResTpOut;

  -- synthesisable functions.
  FUNCTION incr_mod(i,max : IN integer) RETURN integer;
  FUNCTION my_mod(a, b: IN integer) RETURN integer;
END types;

PACKAGE BODY types IS
  FUNCTION incr_mod(i,max : IN integer) RETURN integer IS
    VARIABLE res : integer;
  BEGIN
    IF i < max-1 THEN
      res:=i+1;
    ELSE
      res:=0;
    END IF;
    RETURN res;
  END incr_mod;

  FUNCTION my_mod(a, b: IN integer) RETURN integer IS
  BEGIN
    IF a-b < 0 THEN
     RETURN a;
    ELSE
     RETURN (a - ((a/b) * b)); -- b is a poower of 2
  END my_mod;
END types;
```

Figure 9. The data and bus types

results in Fig.13 were produced. The wave form is a simulation of the scheduler using three gcd resources. From the simulation set we can see that the active signal of a resource directly follows the *wr* signal (This can be seen at clock value $150ns$) and that consecutive writes to the same resource are handled by the system without any extra delays. Which indicates that we do not lose extra clock cycles for writing the data to and form the resource. This scheduler was also synthesized for a 0.5μ technology using the $LeonardoSpectrum$ synthesis engine from Exemplar Logic. Preliminary synthesis results[3]indicated that we were able to obtain a clock frequency of $113.4Mhz$.

---

[3]For synthesis we used the *my_mod* function, see Fig.9, since the *mod* function provided by the language is not synthesisable.

```
ENTITY scheduler IS
  PORT (clk     : IN std_logic;
        reset   : IN std_logic;
        data    : IN DataTpIn;
        Sched2Res : OUT DataSchedResVecTpIn;
        Res2Sched : IN DataSchedResVecTpOut;
        result  : OUT DataTpOut
        );
END scheduler;

ARCHITECTURE behaviour OF scheduler IS
  SIGNAL Sched2Res_int : DataSchedResVecTpIn;
  SIGNAL free, store : std_logic_vector(0 TO Nres-1);
BEGIN
  PROCESS(Res2Sched,Sched2Res_int,free)
  BEGIN
    store <= (OTHERS=>'0');
    FOR i IN Res2Sched'RANGE LOOP
      IF NOT Res2Sched(i).active THEN
        store(i)<=Sched2Res_int(i).wr OR NOT free(i);
      END IF;
    END LOOP;
  END PROCESS;

  PROCESS(clk, reset)
    VARIABLE first, last : integer RANGE 0 TO M-1;
    VARIABLE memory : MemoryTp;
    VARIABLE last_handled : boolean;
    TYPE LutTp IS ARRAY (0 TO Nres-1) OF integer RANGE 0 TO M-1;
    -- stores the index in memory of the result.
    VARIABLE Lut : LutTp;
  BEGIN
    IF reset='1' THEN
      last:=0; first:=0;
      Sched2Res_int <= (OTHERS => ((0,0),'0'));
      free <= (OTHERS => '1');
      last_handled:=false;
      lut := (OTHERS => 0);
    ELSIF rising_edge(clk) THEN
    result <= memory(first).d1;
    -- store valid data from resources to memory
      FOR i IN 0 TO Nres-1 LOOP
        IF store(i)='1' THEN
          -- memory.d1 is used for both input and output data
          memory(Lut(i)).d1:=Res2Sched(i).Data;
          free(i)<='1';
        END IF;
      END LOOP;
      -- assign data to resources
      Sched2Res_int <= (OTHERS => ((0,0),'0'));
      memory(first):=data;
      last_handled:=false;
      FOR i IN free'RANGE LOOP
        IF (free(i)='1' OR store(i)='1') AND NOT last_handled THEN
          -- If resource is free ('0') then it can be used again.
          -- But if store is '1' then the resource has just
          -- finished a job and can allocated to a new one.
          Sched2Res_int(i) <= (memory(last),'1');
          Lut(i) := last;
          free(i)<='0';
          last_handled:=last=first;
          last := incr_mod(last, M);
        END IF;
      END LOOP;
      first := incr_mod(first, M);
    END IF;
  END PROCESS;
  Sched2Res <= Sched2Res_int;
END behaviour;
```

Figure 10. A resource independent scheduler implementation

7 Conclusions

In this paper, the design and implementation of a hard-ware dynamic scheduler for non-manifest loops was described. By letting the input data and the output data share the same memory buffer, the total memory space required by the scheduling system was reduced to a single memory buffer. The memory buffer has a length which is equivalent

```
ENTITY system IS
  PORT (clk    : IN std_logic;
        reset  : IN std_logic;
        data   : IN DataTpIn;
        result : OUT DataTpOut);
END system;

ARCHITECTURE structure OF system IS
  component scheduler IS
    PORT (clk     : IN std_logic;
          reset   : IN std_logic;
          data    : IN DataTpIn;
          Sched2Res : OUT DataSchedResVecTpIn;
          Res2Sched : IN DataSchedResVecTpOut;
          result  : OUT DataTpOut
          );
  END component;
  component resource is
    port(DataIn  : DataSchedResTpIn;
         DataOut : out DataSchedResTpOut;
         clk     : std_logic;
         reset   : std_logic);
  end component;
  SIGNAL Sched2Res : DataSchedResVecTpIn;
  SIGNAL Res2Sched : DataSchedResVecTpOut;
BEGIN
  sched:scheduler
      PORT MAP (clk,reset,data,Sched2Res,Res2Sched,result);
  resources:FOR i IN Sched2Res'RANGE GENERATE
  inst:resource
      PORT MAP (Sched2Res(i),Res2Sched(i),clk,reset);
  END GENERATE;
END structure;
```

Figure 11. The scheduler configuration

```
entity resource is
  port(DataIn  : DataSchedResTpIn;
       DataOut : out DataSchedResTpOut;
       clk     : std_logic;
       reset   : std_logic);
end resource;

architecture test of resource is
  alias x : integer is DataIn.Data.d1;
  alias y : integer is DataIn.Data.d2;
  alias wr : std_logic is DataIn.wr;
  alias z : integer is DataOut.data;
  alias active : boolean is DataOut.active;
  signal active_i : boolean;
  signal xreg,yreg,ixreg,iyreg,zi,xi,yi : int0tomax;
  function module(x,y : integer) return integer is
  begin
    if y=0 then
      report "right operand 0" severity note;
      return x;
    else
      return x mod y; -- for simulation
      -- return my_mod(x,y); -- for synthesis
    end if;
  end module;
begin
  process(clk,reset)
  begin
    if rising_edge(clk) then
      if active_i then
        xreg <= ixreg;
        yreg <= iyreg;
      end if;
    end if;
  end process;
  xi <= x when wr='1' else xreg;
  yi <= y when wr='1' else yreg;
  zi <= module(yi,xi);
  ixreg <= zi;
  iyreg <= xi;
  active_i <= not(zi=0);
  active <= active_i;
  z <= xi;
end test;
```

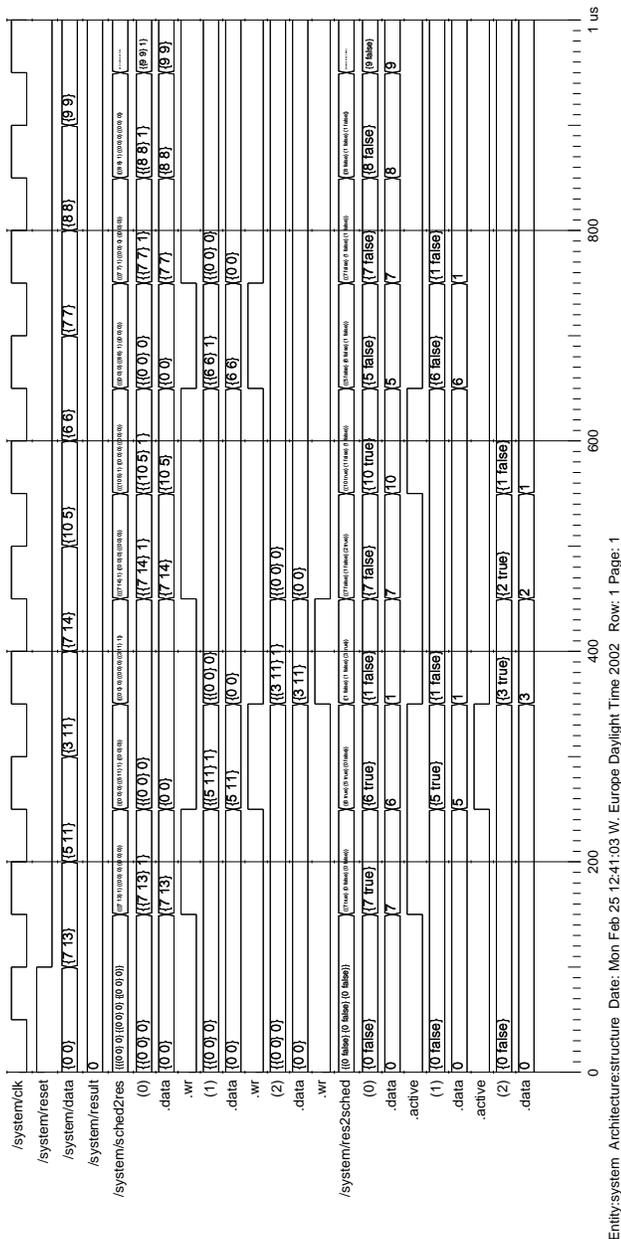**Figure 12. The gcd(x,y)resource implementation**

Figure 13. Simulation results

be performed in one clock cycle will not have extra clock delays for writing and reading the data. Although the number of clock cycles is reduced the length of the critical path can be longer. Current synthesis tools will normally select a clock frequency such that the clock period is longer than the longest critical path within the system. The hardware scheduler presented in this paper allows us to build execution units which are capable of handling an input data stream with known throughput. If the computational load of the window is known before hand we can calculate the exact number of resources that are needed and the latency of the system using equation 3.

8 Future work

In high-throughput applications loop computations with data dependencies among various computations are also plausible. It is usually the case that manifest and non-manifest computations with and without dependencies amongst them, co-exist in one implementation. In order to synthesize a scheduler which is capable of handling all kinds of different programs, the scheduler must be able to cope with data dependencies amongst different computations and still ensure efficient use of the resources within the system.

References

[1] O.Mansour, S.Etalle, T.Krol, "Scheduling and Allocation of Non-Manifest Loops on Hardware Graph Models", PROGRESS Proceedings, 2001, ISBN 90-73461-26-x

[2] W. Verhaegh, "Multidimensional Periodic Scheduling", Ph.D Thesis, University of Eindhoven, The Netherlands, 1995, ISBN 90-74445-21-7

[3] Silvia M. Muller, "On the Scheduling of Variable Latency Functional Units", 11th ACM Symposium on Parallel Algorithms and Architectures SPAA'99

[4] D. Gajski, N.Dutt, A. Wu, S. Lin, "High-level synthesis: Introduction to chip and system design", Kluwer, ISBN 0-7923-9194-2, 1992

[5] ModelSim from Model technology, www.model.com

[6] Vijay Raghunathan, Srivaths Ravi, Ganesh Lakshminarayana, "Integrating Variable-Latency Components into High-Level Synthesis", IEEE Transactions on computer-aided design of integrated circuits and systems, October 2000

[7] L. Benini, E.Macii, M.Poncino, and G.De Micheli, "Telescopic units: A new Paradigm for performance optimization of VLSI designs" IEEE, Trans. Computer-Aided Design of Integrated Circuits and Systems, vol. 19, No. 10, October 2000

to the calculated latency of the system m in equation 3. In a synchronous communication between the scheduler and its resources, 2 extra clock cycles are required for writing the data from the scheduler to a resource and reading the result from the resource back again. By letting the scheduler have a Moore machine implementation and the resource a Mealy machine implementation the communication path between the scheduler and the resource only has one register and the rest is combinatorial logic. Hence computations which can