Description-level Optimisation of Synthesisable Asynchronous Circuits

Luis A. Tarazona, Doug A. Edwards, Andrew Bardsley and Luis A. Plana
Advanced Processor Technologies Group, School of Computer Science
The University of Manchester, Manchester, M13 9PL, United Kingdom
Email: {tarazonl,doug,bardsley,plana}@cs.man.ac.uk

***Abstract*—The syntax-directed synthesis paradigm has shown to be a powerful synthesis approach. However, its control-driven nature results in significant performance overhead. Some methods to reduce this overhead include peephole optimisations, control resynthesis and component optimisations. This work explores new methods of improving the performance of syntax-directed synthesised asynchronous circuits, using the Balsa synthesis system as the research framework. This includes investigating description styles and the usage of language constructs that exploit the directness of the synthesis method to obtain more concurrent and faster circuits. The techniques and optimisations presented here has been tested in a set of non-trivial examples including a 32-bit processor, a Viterbi decoder, and a channel-sliced wormhole router.**

I. Introduction

The syntax-directed synthesis paradigm has shown to be a powerful synthesis approach. However, its control-driven nature, results in significant performance overhead [1]. In an attempt to reduce this overhead, the following circuit-level approaches have been previously reported:

- *Peephole optimisations*: this technique is based on the identification of a pattern of components that can be replaced with an faster alternative [2]–[4].
- *Control resynthesis*: this technique consist on clustering sections of control trees and replacing these with an optimised controller that implements the same behaviour [4], [5].
- *Component optimisation*: this is based on finding alternative designs for the handshake components that result in more concurrent, faster operation [6].

An orthogonal alternative to the above is to exploit the *directness* of the synthesis method at the description level. Highly expressive, high-level description languages like Balsa and Haste [7] can result in naïve descriptions with poor performance unless the designer has a good understanding of the underlying compilation process. Furthermore, it is often claimed that in this approach, an experienced designer could make performance/power/area trade-offs. This task would be easier if the designer could have some insight into the impact of a particular construct or coding style.

This work explores the effects of directness in the performance of Balsa synthesised circuits and proposes coding techniques and optimisations that result in more concurrent, faster implementations. The techniques and optimisations presented here has been tested in a set of non-trivial examples including a 32-bit processor, a Viterbi decoder, and a channel-sliced wormhole router.

This paper is organised as follows: Section II introduces syntax-directed synthesis and the Balsa synthesis system. Section III summarises the related work. Section IV introduces the data-driven description style as an efficient way of describing circuits in Balsa/Haste. Section V introduces the description-level optimisations proposed in this work. Section VI presents simulation results for the examples listed above. finally, Section VII summarises this work and proposes future work.

II. Syntax-directed synthesis

The syntax-directed approach to synthesise asynchronous circuits is based in the compilation of descriptions written in a high-level language into a communicating network of predesigned modules. The compilation process performs a one-to-one mapping of each language construct into the network of components that implements it. This transparent mapping gives a high degree of flexibility in the design as incremental changes to the specification generates predictable changes in the resulting circuit, allowing the designer to optimise the circuit in terms of performance, power or area, at the description language level. The compiled network of handshake components constitutes an intermediate representation that can be subsequently replaced by a gate netlist.

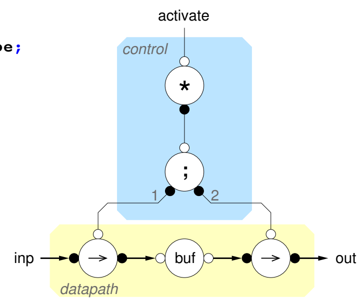
Currently there exist two fully automated CAD systems that use this approach for the synthesis of asynchronous systems: *Haste* (formerly called *Tangram*) [7] and *Balsa* [8], an open-source system developed at the University of Manchester that closely follows the Tangram philosophy. Syntax-directed synthesis has been used successfully in the synthesis of several VLSI systems, including the SPA processor [9], and the ARM996HS [10].

A. The Balsa synthesis system

Balsa is the name for both the framework for synthesising asynchronous circuits and the language used to describe such systems. Balsa uses the syntax-directed synthesis approach to generate *handshake circuits* from a description written in the Balsa language. Originally introduced by van Berkel [11], a handshake circuit is a communicating network of handshake components connected point-to-point using *handshake channels*. Each channel connects exactly one passive port of a handshake component to an active port of another handshake component. As an example, consider the Balsa specification for a simple 1-place buffer (register) shown in figure 1(a). The specification is parameterised in the type of data the register can hold. The register has an input channel `in` and an output channel `out`. The variable `buf` stores the data and the operation consists of an unbounded repetition (`loop`) of two actions: input data (->) from channel `in` into `buf` sequenced (`;`) with output (<-) of the data stored in `buf` to channel `out`. Figure 1(b) shows the handshake circuit generated by Balsa from the code at its left, where the *Loop* component is labelled with a star ($*$).

```
procedure buffer
(
  parameter DataType : type;
  input inp   : DataType;
  output oot  : DataType
) is
  variable v : Datatype
begin
    loop
        inp -> v
        ;
        out <- v
    end
end
```

(*a*)



(*b*)

Figure 1: Balsa 1-place buffer.

III. Related work

Balsa has previously been used to demonstrate the impact on performance of some description-level techniques combined with the introduction of more concurrent handshake components. In particular, true asynchronous operation of the system pipeline, a data-driven coding style are presented as performance-driven description techniques [6].

In a recent work, Hansen and Singh [12] describe a series of automated “source-to-source” transformations that optimise syntax-directed descriptions using a variety of concurrency-enhancing optimisations including: automatic parallelisation, automatic pipelining, arithmetic optimisation and reordering of channel communication. Although considerable speed-ups are claimed, some of the example designs start with extremely naïve code sequences, where significant improvements can be easily obtained. Also, their proposed approach is limited to *slack elastic* [13] systems descriptions only (a slack elastic system preserves correct operation even if extra pipeline buffer stages are introduced in any channel). This limitation reduces the usefulness of an “automated” approach as it is frequently necessary for the designer to understand the nature of the transformations to ensure they are safe, which may represent a considerable design effort for the user.

The approach used here is more general and attempts to give the designer a clearer understanding of the source of performance inefficiencies, the techniques available to reduce it and the trade-offs made. As an additional and important benefit, manual optimisation techniques can be applied to exploit the designer’s knowledge about the behaviour of the system. This knowledge is something that is more complex to automate because it cannot be inferred by analysing the code. This work is complementary to the approaches presented above and to the circuit-level optimisation techniques. The techniques presented here could also serve as a source for optimising compilers or to enhance automated source-to-source transformations.

IV. The data-driven description style

In Balsa/Haste it is relatively easy for an user to write a working, but most likely low-performance, description of a system due to their similarities with C and Verilog language. One of the major challenges for an asynchronous designer is to learn to think in terms of concurrent processes, instead of the easier to understand sequential processing found in imperative languages. An imperative, sequential description generates a large control tree that directs the flow of data in the datapath. This large control tree results in performance penalties that tends to increase with the complexity of the description.

However, it is possible to describe a more concurrent operation by using a *data-driven* description style, that is, a description in which the arrival of data activates the units. In this style the description of a circuit is divided into simpler, concurrent actions that communicate using channels. Given the asynchronous nature of the circuits, these actions are activated immediately by the data arriving at their inputs, process the information and generate outputs to activate the next unit. The resulting control tree is generally small and local to the modules implementing the actions.

Key to implementing data-driven circuits is an adequate partitioning of the circuit into groups of actions that source and consume data. Internal channels will connect theseactions. The partitioning also involves determining the group of actions that will necessarily require sequencing, as unnecessary sequencing is a well-known source of overheads. Sequencing is normally associated with the use of variables but also may be required to prevent deadlocks. Every variable that has a write-then-read access pattern inside each iteration of a group of actions can be substituted by a channel write and an enclosing read (where the value can be read as many times as required). Only variables that store a value required in the next iteration need to be left in the description.

V. Optimising data-driven descriptions

In this section different description techniques will be introduced in order to achieve the goal of writing performance-optimised data-driven descriptions. To give a clearer idea of the effects in the code, a simplified version of the Branch Metric Unit (BMU) of an asynchronous Viterbi decoder will be used here as a running example.

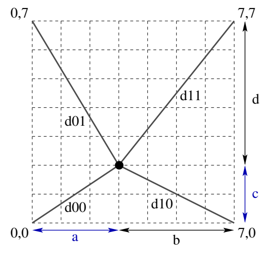
A. The Branch Metric Unit example

Consider the description of a branch metric unit (BMU) for a soft-decision-based asynchronous Viterbi decoder [14]. This unit takes two 3-bit quantities (a,c) which are soft-coded representations of the two received bits in a Viterbi decoder. For each input, 000 (0) denotes the reception of a strong zero and 111 (7) indicates a strong 1.

The task of the BMU is to calculate the distance (branch weight) between the received pair and the ideal branch pattern symbols (0,0), (0,7), (7,0), (7,7), as shown in figure 2(a). The distance to be calculated is the Manhattan distance, as this turns out to be equivalent to the Euclidean distance squared in this application [15]. The required branch weights are: $d00 = a + c$, $d01 = a + d$, $d10 = b + c$, $d11 = b + d$, where $b = 7 - a$ and $d = 7 - c$. Figure 2(b) depicts the BMU algorithm. The data-driven description of the simplified BMU is shown in figure 3(a).

```
input (a,c) ;
b ← 7 - a ;
d ← 7 - c ;
d00 ← a + c ;
d01 ← a + d ;
d10 ← b + c ;
d11 ← b + d ;
output(d00, d01, d10, d11)
```

(a)



(b)

Figure 2: Branch metric computation for a Viterbi decoder [14].

B. Separating actions into concurrent loops

The example code in figure 3(a), which is already split in two groups of actions, can be split into two concurrent enclosed groups instead of having two nested enclosures. Furthermore, the outer unbounded loop can be split into two concurrent unbounded loops, where any value of the original enclosure required in the second loop must be passed using new internal channels. In this example, the values of `ia` and `ic` required in the second group are transferred together with `b` and `d`, as shown in figure 3(b). In general, this "splitting" can continue until all grouping possibilities are exhausted, according to the dependencies of the commands.

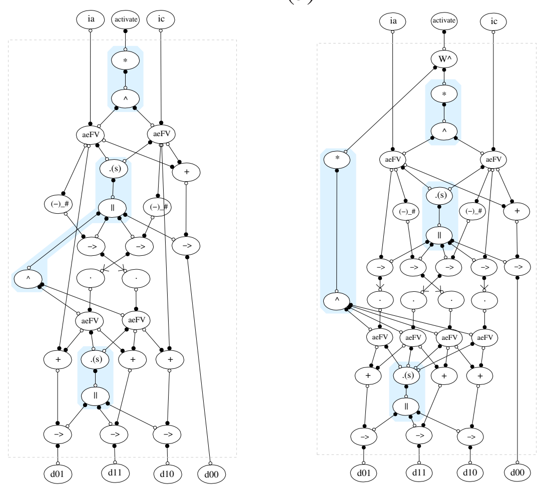
The resulting circuits for the original and with the splitted loops are shown in figure 3(c) and (d), respectively. After the splitting process the datapath will be a pipelineable description *without* pipeline registers. On the control side, the control tree in the middle has been split and now the control for the second round of computations runs concurrently with the control of the input section. The new description results in the addition of two extra *aeFV*s (for the copies of `ia` and `ib` passed to the bottom loop). The four *aeFV*s decouple the RTZ phases of the control of the two loops, without adding any latency.

```
loop
    ia, ic ->! then -- read inputs                                  (1)
        -- first batch of calculations                              (2)
        b <- (7 - ia as TOut) || d <- (7 - ic as TOut) ||
        d00 <- (ia + ic as TOut) ||
        -- compute the other metrics                                (3)
        b, d ->! then
            d01 <- (ia + d as TOut) ||
            d10 <- (b + ic as TOut) || d11 <- (b + d as TOut)
    end
end
```

(a)

```
loop
    ia, ic ->! then
        b   <- (7 - ia as TOut)  || d <- (7 - ic as TOut) ||
        d00 <- (ia + ic as TOut) || ta  <- ia || tc  <- ic
    end
end ||
loop -- compute the other metrics
    ta, tc, b, d ->! then
        d01 <- (ta + d as TOut) ||
        d10 <- (b + tc as TOut) || d11 <- (b + d as TOut)
    end
end
```

(b)



(c) (d)

Figure 3: The simplified BMU Balsa description: (*a, c*) original, (*b, d*) with separated loops.

The results for the BMU description that uses this technique is labelled *"Lopt eager"* in the graphs of figure 4. All results are normalised to those of the original BMU design presented earlier. Let us refer for now to the first group of bars labelled *"no ch. broadcast"* in figure 4 (the other groups of results will be introduced later). From the graphs, the performance gain using the technique just introduced is ∼1.5 with a relative area and energy of ∼1.3 and ∼1.5 respectively.

An important remark with respect to the level of granularity of this technique is that the throughput will depend on the slowest stage and increasing the pipeline depth will increase the latency. Indiscriminate loop splitting (either manually or automatically) by just analysing precedences and/or dependencies may end up being suboptimal. The designer must take into account the balancing of the pipeline, the nature of the data and the behaviour of the environment among other factors. Being able to express the designer's knowledge about the circuit is an advantage but also a challenge in syntax-directed descriptions.

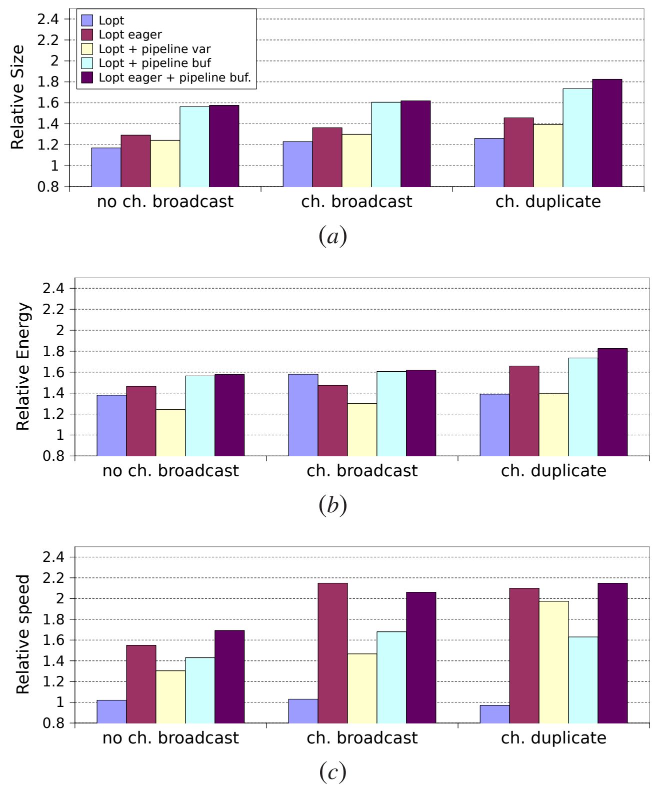


Figure 4: Simulation results of different optimisations applied to the BMU.

C. Broadcasting values

Often within a pipeline, a value from a channel is required unconditionally and concurrently by more than one stage in the pipeline, as noticed previously with `ia` and `ic`. Enclosure provides a means for multicasting values but it may prevent finer grain concurrency and deeper pipelining.

For instance, in the code of figure 3(a) the groups of actions (2) and (3) are within the same enclosure, hence no new token can be processed by action (2) until action (3) has finished. A solution for this, shown previously in the loop splitting example (figure 3(b)), relied on duplicating the values required by the next group of actions inside the active enclosure, but more concurrent solutions for broadcasting are possible. In Balsa, there are two ways of specifying multiple concurrent receivers for the same channel:

i. Using implicit *broadcasting*: In the description, the channel is read in every place that it is required. In this case, the reads are fully synchronised: the data will be available to the reading processes only after every read request has been received. Similarly, data withdrawal will begin only after all reading processes have signalled the consumption of data.
ii. Using explicit *duplication* of the channel by means of enclosure. This method provides more decoupling between processing the RTZ phases of the reads, as every request will be granted independently of the arrival of the others.

```
loop
    ia, ic ->! then
        b <- (7 - ia as TOut) || d <- (7 - ic as TOut)
    end
end ||
-- ia and ic reuse in next loop
-- creates implicit broadcasting
loop
    ia, ic, b, d ->! then
        d00 <- (ia + ic as TOut)|| d01 <- (ia + d as TOut) ||
        d10 <- (ic + b as TOut) || d11 <- (b + d as TOut)
    end
end
```

(*a*)

```
loop -- make two copies of ia explicitly
    ia ->! then  a1 <- ia || a2 <- ia end
end ||
loop -- make two copies of ic explicitly
    ic ->! then c1 <- ic || c2 <- ic end
end  ||
loop
    a1, c1 ->! then
        b <- (7 - a1 as TOut) || d <- (7 - c1 as TOut)
    end
end ||
loop
    a2, c2, b, d ->! then
        d00 <- (ta + tc as TOut)|| d01 <- (ta + d as TOut) ||
        d10 <- (tc + b as TOut) || d11 <- (b + d as TOut)
    end
end
```

(*b*)

Figure 5: Broadcasting: (*a*) Implicit broadcasting. (*b*) Explicit duplication.

The code in figure 5 show these two forms of broadcasting in the simplified BMU example. This technique further improves concurrency, which results in higher performance at the cost of some area and energy penalties. The bins labelled *"ch. duplicate"* and *"ch. broadcast"* in the graphs of figure 4 shows the results for the complete BMU design when these techniques are applied. Referring to the *"Lopt eager"* columns, the relative speed is now ∼2.1 (slightly larger for the broadcast method). The relative area and energy are ∼1.45 and ∼1.65 when using channel duplication and a bit smaller (∼1.35 and ∼1.50) when using implicit broadcasting.

In this particular example, the synchronisation penalty imposed by the implicit broadcasting is not apparent because the design has balanced threads: all four outputs are generated using similar operations and the simulation environment generate inputs and consumes outputs eagerly. In designs with this balanced behaviour, broadcasting has the advantage of less area and energy penalties. However, in designs with more complex, imbalanced thread execution patterns like a processor, thread decoupling provided by explicit duplication allows a head start for some of the threads required to complete an instruction, resulting in fully asynchronous operations and better performance.

In common with the previous technique, it is difficult to predict the places or levels of granularity to apply efficiently this technique by only analysing the operations precedence or data dependencies without input from the designer's knowledge about the system.

D. Adding pipeline registers

To increase its throughput, a pipelined description requires inter-stage pipeline registers to decouple them. These can be added in two ways:

i. Using *pipeline variables* within the stage instead of the active enclosure, as presented in [12].
ii. Using explicit pipeline buffer modules (like the one described in section II-A) between stages.

These two styles are shown in the example codes of figure 6. Use of pipeline variables adds a *Sequencer* to the control tree and results in lower performance than the use of explicit pipeline buffers. Results in the graphs of figure 4 reveal this performance penalty. However, pipelining using variables is cheaper in terms of area and energy because no extra *FalseVariable* and *Passivator* components are required.

```
-- Pipeline variables :
-- va, vc, vta, vtc, vb, vc
loop
    [ ia -> va || ic -> vc ] ;
    [ b <- (7 - va as TOut) || d <- (7 - vc as TOut) ||
      ta <- va || tc <- vc ]
end ||
loop
    [ ta -> vta || tc -> vtc || b -> vb || d -> vd ] ;
    [ d00 <- (vta + vtc as TOut)|| d01 <- (vta + vd as TOut) ||
      d10 <- (vtc + vb as TOut) || d11 <- (vb + vd as TOut) ]
end
```

(*a*)

```
-- procedure buf3 is buffer(TInp)
-- procedure buf4 is buffer(TOut)
buf3(a, pa) || buf3(c, pc) ||
loop
    pa, pc ->! then
        b <- (7 - pa as TOut) ||
        d <- (7 - pc as TOut) || ta <- pa || tc <- pc
    end
end ||
buf3(ta, pta) || buf3(tc, ptc) ||
buf4(b, pb) || buf4(d, pd) ||
loop
    pta, ptc, pb, pd ->! then
        d00 <- (pta + ptc as TOut)|| d01 <- (pta + pd as TOut) ||
        d10 <- (ptc + pb as TOut) || d11 <- (pb + pd as TOut)
    end
end
```

(*b*)

Figure 6: Pipelining: (*a*) using variables. (*b*) using explicit pipeline buffers.

Results for the design that uses pipeline variables are labelled *"Lopt + pipeline var"*. Results for the designs that use explicit buffering are labelled *"Lopt + pipeline buf."* and *"Lopt eager + pipeline buf."* (with active eager inputs). Notice how in this case, the synchronisation imposed by channel broadcasting has limited the effectiveness of the decoupling.

A detailed look at the results in figure 4 reveals that adding pipeline registers when using broadcasting or channel duplication has not noticeably increased the performance, but has increased the area and energy penalties. There are two reasons for this: Firstly, the BMU stages are very simple and have low latency (four bit adders/comparators), the extra latency of the pipeline registers reduces their possible benefits. Secondly, as seen in the previous examples (figure 3), the use of active inputs requires *PassivatorPush* components to interface with active outputs.

If dual-rail or other DI data encoding is used, the *PassivatorPush* components require storage in the form of C-elements as shown in figure 7(b). Hence, the *PassivatorPush* acts as a simple "half-latch" [15], [16] that allows the active output to withdraw the data (after synchronising with the active input request) while the other side is in the processing phase. Each time a channel is duplicated using active enclosure, a half-latch is added to the pipeline, providing decoupling between stages. Inserting explicit pipeline registers in this case will only contribute to increase the latency and area of the circuit.

In summary, the implicit storage added to the channels when specifying active inputs serves in some cases as a pipeline register which, when combined with the optimised control of the active eager inputs, efficiently implements decoupling between pipeline stages.

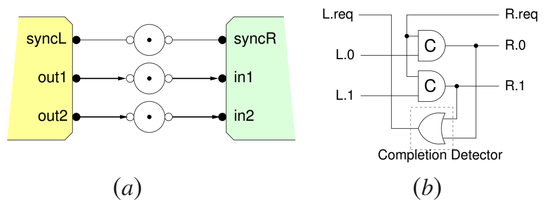


Figure 7: (*a*) Interfacing of two Balsa modules using *Passivators*. (*b*) A 1-bit dual-rail *PassivatorPush*.

E. Optimising guards

Another common source of inefficiencies when coding in Balsa is related to the implementation of the guard expressions for conditional loops and for the `case` and `if` constructs. These conditional constructs require the use of handshake circuits that generate control channels from the datapath. In many cases, the designer can optimise these datapath-generated control by evaluating the guards before their use in the construct, as will be demonstrated here.

Consider the GCD algorithm example, that computes the greatest common divisor of an integer. Figure 8 shows a specification of the GCD algorithm. Figure 9(a) shows a

```
input (a, b);
while a ≠ b do
    if a > b  then a ← a − b;
              else b ← b − a;
output (a);
```

Figure 8: A pseudo-code specification of GCD [15].

GCD device	$t_{cycle}(ns)$	Relative speed	Area (transistors)	Relative area	Relative energy
Original	181.68	1.00	6856	1.00	1.00
Optimised	133.26	1.36	6991	1.02	1.14

Table I: GCD Simulation results.

direct implementation of the algorithm in Balsa. In the implementation, the two guards (`va /= vb` and `va > vb`) are evaluated only after the control reaches each conditional structure, resulting in an unnecessary delay. The code also exhibits the common "problem" of auto-assignment, which in most cases introduces additional performance penalties.

The performance-optimised description of the GCD shown in figure 9(b) illustrates how to solve the above problems: Firstly, to avoid auto-assignment, two additional variables (`tva` and `tvb`) are used as temporary storage. Secondly, the two required guards are evaluated in parallel and stored using 1-bit variables `neq` and `gt`. The resulting handshake circuits are shown below the code.

Notice in the circuit at the left how the body of the `loop ... while` (highlighted) contains four sequenced operations:

i. Evaluate the guard expression for the `loop ... while` construct and proceed accordingly.
ii. Evaluate the guard expression for the `if` construct and make the decision.
iii. Update one of the auxiliary variables (labelled only for variable `b` in the circuit).
iv. Update one of the variables (labelled only for variable `b` in the circuit).

In the optimised circuit at the right the loop has only only three sequenced operations:

i. Read the guard expression for the `loop ... while` construct and proceed accordingly.
ii. Read the guard expressions for the `if` construct *and* update one of the auxiliary variables.
iii. Evaluate and store both guards, and update both variables.

Table I shows the simulation results for the two circuits above. The table compares the average time required to calculate the GCD of two 8-bit numbers, area and energy. As the reader may have already noticed, in this example area and energy are being traded for speed: on each iteration, there is a redundant update operation on the variable that does not change and two 1-bit variables are used. The design with the optimised guard is 36% faster at the cost of 14% extra energy and negligible area increase.

F. Encoding multiple guards

In situations where multiple guards are required, it is better to encode the guards into a multi-bit variable and use a `case` construct instead of the more straightforward (but slow) multi-guarded `if` construct. Consider the example code in figure 10 adapted from the description of the input buffer of a *sliced-channel* wormhole router designed in Balsa [17]. Each router has five I/O ports, namely, *Local*, *North*, *South*, *East* and *West*. The code shown corresponds to the *South* input buffer and has been simplified for clarity: only the operations over the dataless *sync* channels that generate the request to the destination ports are detailed.

```
begin
  loop
    d_in[0] -> buf[0];
    -- NOTE: buf[0][4..7] = X, buf[0][0..3] = Y
    if (#(buf[0])[4..7] as 4 bits) < addrX then sync req[NORTH]
        -- data transfer commands omitted
    |   (#(buf[0])[0..3] as 4 bits) > addrY then sync req[EAST]
        -- data transfer commands omitted
    |   (#(buf[0])[0..3] as 4 bits) < addrY then sync req[WEST]
        -- data transfer commands omitted
    else sync req[LOCAL]
    -- data transfer commands omitted
    end
  end
end
```

Figure 10: Simplified description of the *South* input buffer of a sliced-channel wormhole router [17].

The first value received at input `d_in[0]` is the *header flit*. It contains the XY destination addresses that will be compared with the addresses of the router `addrX` and `addrY`. The destination is chosen accordingly to the comparisons and the order of priority specified in the description.

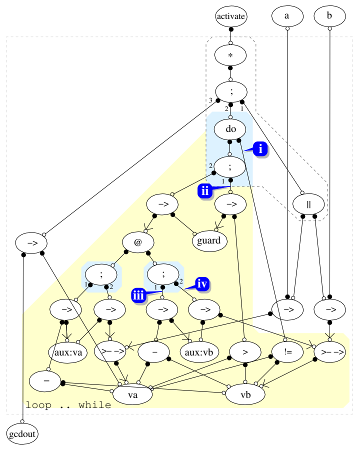
```
begin
  loop
  -- NOTE: d_in[0][4..7] = X, d_in[0][0..3] = Y
    d_in[0] ->! then
      n <- (#(d_in[0])[4..7] as 4 bits) < addrX ||
      e <- (#(d_in[0])[0..3] as 4 bits) > addrY ||
      w <- (#(d_in[0])[0..3] as 4 bits) < addrY
      d_in0 <- d_in[0] -- replicate d_in required
    end
  end ||
  loop
    n, e, w ->! then
      case (#w @ #e @ #n as 3 bits) of
       0b1xx then sync req[NORTH]
        -- data transfer commands omitted
      |0b01x then  sync req[EAST]
        -- data transfer commands omitted
      |0b001 then  sync req[WEST]
        -- data transfer commands omitted
      else sync req[LOCAL]
        -- data transfer commands omitted
      end
    end
  end
end
```

Figure 11: Optimised, simplified description of the *South* input buffer of a sliced-channel wormhole router [17].

The optimised code is shown in figure 11. In this new

```
type dtype is 8 bits
procedure gcd
(
  input a, b : dtype;
  output gcdout : dtype
) is
  variable va, vb : dtype
begin
   loop
      [ a -> va || b -> vb ] ;
      loop
      while va /= vb then
         if va > vb then  va := (va - vb as dtype)
         else vb := (vb - va as dtype)
         end
      end ;
      gcdout <- va
   end
end
```

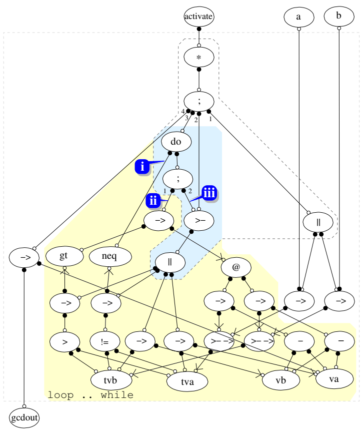
(*a*)



(*c*)

```
type dtype is 8 bits
procedure gcd
(
  input a, b : dtype;
  output gcdout : dtype
) is
  variable va, vb, tva, tvb : dtype
  variable neq, gt : bit
begin
   loop
      [ a -> tva || b -> tvb ] ;
      loop
         neq := tva /= tvb || gt := tva > tvb ||
         va := tva || vb := tvb
      while neq then
         if gt then tva := (va - vb as dtype)
         else tvb := (vb - va as dtype)
         end
      end ;
      gcdout <- va
   end
end
```

(*b*)



(*d*)

Figure 9: Two implementations of the GCD algorithm in Balsa and their compiled handshake circuits.

description, instead of using the `if` construct, all guards are evaluated and stored in parallel with the buffering of the input value within an active enclosure. The four bits generated by these evaluation are then joined and used as the guard expression of a `case` construct. Also, in this new construct the encoding of the guards reflect the priority expressed in the original description.

VI. Evaluation of Design Examples

The optimisations presented here were evaluated using the following design examples: (i) the 32-bit nanoSpa processor core [6], (ii) a Viterbi decoder (VD), (iii) a channel-sliced wormhole router [17]. All results were obtained for pre-layout, transistor-level simulations, using a 180nm technology cell library.

Table II shows the simulation results for the nanoSpa design. *DD* is the data-driven reference design. the table shows that adding the separation of actions into concurrent loops and explicit broadcasting (*+BLopt*) increases the performance in 16.45%. Adding the guard optimisations, the improvement reaches 23.45%. In both cases, Area and energy penalties/improvements are negligible.

nanoSpa	*DMIPS*		*Area*		*Energy*	
device	*absolute*	*Δ (%)*	*elements*	*ratio*	*μJ*	*ratio*
DD (ref)	63.15	—	622884	1.00	0.358	1.00
+BLopt	73.54	16.45	662734	1.06	0.361	1.01
+Gopt	77.96	23.45	611793	0.98	0.355	0.99

Table II: NanoSpa simulation results.

Table III summarises the results for the Viterbi decoder. The parameter used to measure the performance is the average output data rate. The design VD(ref) is the original unoptimised description. VDO is the description-level fully-optimised version VDO. In this particular design, the performance obtained by optimising guards was negligible and is not included in the results.

The results indicate that the description-level optimised design achieves more than twice the speed of the original description. It is worth comparing this result with the 23% obtained with the more complex nanoSpa description. There

Decoder	*data rate*		*Area*		*Energy*	
device	*Msps*	*Δ (%)*	*elements*	*ratio*	*μJ*	*ratio*
VD	31.59	—	58815	1.00	0.145	1.00
VDO	64.75	200.5	80640	1.37	0.218	1.50

Table III: Viterbi decoder simulation results.

are two reasons for this difference: firstly, the reference design was not in data-driven style as the nanoSpa and secondly the difference in complexity between the designs makes easier to improve the critical path with the optimisations.

Table IV shows the results for the wormhole router, where WR is the original unoptimised description, *WR+DL* includes the guard optimisation and guard grouping, and *WR+DL+B* includes explicit data broadcasting in the output buffers. The parameter used to measure the performance is the average period of the *flits*.

Router	T_{flit}		*Area*		*Energy*	
device	*ns*	*Δ (%)*	*elements*	*ratio*	*pJ/flit*	*ratio*
WR	1.40	—	103251	1.00	11.64	1.00
WR+DL	1.31	7.3	88762	0.86	12.49	1.07
WR+DL+B	1.20	17.4	117856	1.14	15.40	1.32

Table IV: Balsa wormhole router simulation results.

Results show that the description-level optimisation of guards has increased the performance in 7.3% with a reduction in area to 86% of the original and a penalty of 7% in energy. Applying a more aggressive optimisation in the output buffers, takes the speed-up to 17.4% at the expense of larger area and energy penalties.

VII. Conclusions and future work

This work has presented a number of description-level optimisations together with their effects in performance, resulting circuit structures and trade-offs made. these description level techniques included: separation of actions within unbounded loops to increase concurrency, broadcasting styles and stage decoupling techniques. Early evaluation of guards and encoding of multiple guards for conditional loops and `case` constructs were also presented as a way of increasing the performance. Because the structures that implement the mentioned constructs generate control signals from the datapath, optimising the decision-making circuit speeds up the control. The effects of the use of active eager enclosures with the above techniques were also analysed. The reduced control tree achieved with these optimisation techniques combined with the head start of the control provided by the active eager enclosure contribute to the increase in performance of the circuit. The effects on the performance of the circuits clearly depend on the nature of the operations implemented. However, there usually will be some energy and area penalty as shown in the results.

Most of the optimisations presented here may be automated, with the guard encoding and optimisation possibly requiring a wider window of exploration. However, the optimisations also serve as a guidance to exploit the designer's knowledge of the system behaviour. Future work includes using the circuit structures that result from the optimised descriptions as a reference to create optimised mappings in a optimisation step of the compiler or can be incorporated as rules for automated source-to-source transformation tools.

References

[1] S. Taylor, D. Edwards, and L. Plana, "Data-driven asynchronous circuits," in *Proc. International Symposium on Asynchronous Circuits and Systems*. IEEE Computer Society Press, Apr. 2008, pp. 3–14.

[2] K. van Berkel and M. Rem, "VLSI programming of asynchronous circuits for low power," in *Asynchronous Digital Circuit Design*, G. Birtwistle and A. Davis, Eds. Springer-Verlag, 1995, pp. 152–210.

[3] A. Peeters and K. van Berkel, "Single-rail handshake circuits," in *Proc. Working Conf. on Asynchronous Design Methodologies*, May 1995, pp. 53–62.

[4] T. Chelcea, A. Bardsley, D. Edwards, and S. M. Nowick, "A burst-mode oriented back-end for the Balsa synthesis system," Mar. 2002, pp. 330–337.

[5] T. Kolks, S. Vercauteren, and B. Lin, "Control resynthesis for control-dominated asynchronous designs," in *Proc. International Symposium on Asynchronous Circuits and Systems*, Mar. 1996.

[6] L. Plana, D. Edwards, S. Taylor, L. Tarazona, and A. Bardsley, "Performance-driven syntax directed synthesis of asynchronous processors," in *Proc. International Conference on Compiles, Architecture & Synthesis for Embedded Systems*, Sept. 2007, pp. 43–47.

[7] H. S. company website, *http://www.handshakesolutions.com/Technology/Haste*.

[8] A. Bardsley, "Implementing Balsa handshake circuits," Ph.D. dissertation, Department of Computer Science, University of Manchester, 2000.

[9] L. A. Plana, P. A. Riocreux, W. Bainbridge, A. Bardsley, J. D. Garside, and S. Temple, "SPA – a synthesisable Amulet core for smartcard applications," in *Proc. International Symposium on Asynchronous Circuits and Systems*. IEEE Computer Society Press, Apr. 2002, pp. 201–210.

[10] A. Bink and R. York, "ARM996HS: the first licensable, clockless 32-bit processor core," *IEEE Micro*, vol. 27, no. 2, pp. 58–68, Mar. 2007.

[11] K. van Berkel, *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*. Cambridge University Press, 1993.

[12] J. Hansen and M. Singh, "Concurrency-enhancingtools for validating asynchronous digital circuits," in *Proc. International Symposium on Asynchronous Circuits and Systems*. IEEE Computer Society Press, Nov. 1994, pp. 12–21.

[13] R. Manohar and A. J. Martin, "Slack elasticity in concurrent computing," in *Proc. 4th International Conference on the Mathematics of Program Construction*, ser. Lecture Notes in Computer Science, J. Jeuring, Ed., vol. 1422, 1998, pp. 272–285.

[14] L. E. M. Brackenbury, *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001, ch. An Asynchronous Viterbi Decoder, pp. 240–272.

[15] J. Sparsø and S. Furber, Eds., *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001.

[16] C. Brej, "Early-output logig and anti-tokens," Ph.D. dissertation, Department of Computer Science, University of Manchester, 2005.

[17] W. Song and D. Edwards, "Building asynchronous routers with independent sub-channels," in *Proc. International Symposium on System-on-Chip 2009*, Oct. 2009.