Data path analysis for Dynamic Circuit Specialisation

Tom Davidson
Hardware and Embedded Systems team,
Computer Science Lab
Ghent University
Sint-Pietersnieuwstraat, 41
Ghent, Belgium 9000
Email: tom.davidson@ugent.be

Dirk Stroobandt
Hardware and Embedded Systems team,
Computer Science Lab
Ghent University
Sint-Pietersnieuwstraat, 41
Ghent, Belgium 9000
Email: dirk.stroobandt@ugent.be

***Abstract*—Dynamic Circuit Specialisation (DCS) is a method that exploits the reconfigurability of modern FPGAs to allow the specialisation of FPGA circuits at run-time. Currently, it is only explored as part of Register-transfer level design. However, at the Register-transfer level (RTL), a large part of the design is already locked in. Therefore, maximally exploiting the opportunities of DCS could require a costly redesign. It would be interesting to already have insight in the opportunities for DCS from the higher abstraction level. Moreover, the general design trend in FPGA design is to work on higher abstraction levels and let tool(s) translate this higher level description to RTL. This paper presents the first profiler that, based on the high-level description of an application, estimates the benefits of an implementation using DCS. This allows a designer to determine much earlier in the design cycle whether or not DCS would be interesting. The high-level profiling methodology was implemented and tested on a set of PID designs.**

I. Introduction

Most recent FPGA families have run-time reconfiguration (RTR) capabilities. These capabilities are generally used to switch out large logic functions that are mutually exclusive in time [2]. However, RTR can also be used to implement Dynamic Circuit Specialisation (DCS). DCS allows an FPGA design to be dynamically specialized for a subset of its current input signals. These signals are called parameters. The specialized circuits are smaller, and in some cases faster, than the original circuit.

Determining whether or not an application will benefit from a DCS implementation is difficult for the designer. This requires a trade-off between the benefits and the overhead of DCS. Predicting the benefits of DCS, saved area and/or increased speed, requires very detailed knowledge of the specific way the application is implemented in RTL. Moreover, DCS also introduces an overhead. A new specialized circuit is generated each time any parameter value(s) change. This new circuit is loaded into the FPGA through partial run-time reconfiguration. Both processes, the specialisation and the reconfiguration, introduce a time and resource overhead. The resource overhead is fixed, but the time overhead is incurred each time the parameter changes. Thus, the total time overhead DCS adds to the execution time depends on the dynamic signal behaviour of the chosen parameter. Without any tools, getting a good overview of the benefits and the overhead of DCS almost requires implementing the full DCS design.

In earlier work of Davidson et al. [3], a tool was presented that automatically determines the most interesting parameter candidates in an RTL design and estimates the consequences of implementing a DCS design with each parameter candidate. However, it would be more interesting to identify opportunities for DCS earlier in the design process, as this allows designing the RTL implementation with DCS already in mind. Additionally, the current design trend for FPGAs is clearly moving to a higher abstraction level. One example is the recent incorporation of high level synthesis tools into the Xilinx FPGA flow [1]. We present a profiler that answers both questions, it aims to identify opportunities for DCS from a higher abstraction level.

First, in Section II, we will give an overview of DCS and a tool flow that implements DCS, the TLUT tool flow. This section also gives a more in depth overview of the DCS overhead. Section III presents the high-level profiler. First the profiling methodology is outlined and then the practical implementation is discussed. Experimental results are presented in Section IV and Section V concludes this paper.

II. Dynamic Circuit Specialisation

There are multiple ways to implement a design using DCS. The vendor tools are generally not very useful for implementing DCS, as the specialized circuit needs to either be generated at run-time or be pre-generated and retrieved from memory at run-time. For a small number of possible parameter values this is feasible using the vendor tools. However, as parameter bits increase, pre-generating and storing all possible specialized circuits becomes impossible. An 8-bit parameter already requires storing and pre-generating 2^8 bitstreams. Using the vendor tools to generate the bitstreams at run-time is also impractical as, depending on circuit-size, this can take hours for one specialized circuit. This is shown in the work of Abadei [4].

A. TLUT: a DCS tool flow

Both of these problems were solved by research of the HES group in the Ghent university, in the work of of Bruneel et al. [5], who introduces the concept of parameterized configurations. A *parameterized configuration* (PC) is an FPGA configuration that consists of both static bits and bit-values expressed as multi-valued Boolean functions of the parameters. These Boolean functions are called *tuning functions*. Generating a new specialized configuration from a PC only requires evaluating the tuning functions, using the new parameter values. This evaluation is much faster (μs) than the regular FPGA tool flow and a PC requires much less storage than separate already specialized bitstreams.

The HES group has developed an FPGA flow, the *TLUT tool flow*, that takes a parameterized RTL description at the input and outputs a parameterized configuration. In this tool flow, only the truth tables of specific LUTs contain tuning functions. These LUTs are called Tuning LUTs or TLUTs. At-run time, the truth tables of TLUTs will change according to the parameter values. Since only LUT truth tables are changing, the costly place and route algorithms do not have to be run during the specialisation. Moreover, only LUT truth tables have to be written to the FPGA configuration memory which saves FPGA reconfiguration time. The TLUT tool flow can be integrated with the Xilinx FPGA flow and supports Virtex II Pro, Virtex 5 and 7-Family FPGAs. It is open source and available on github [6].

The benefits of the TLUT tool flow are significant. For example, a 16-tap adaptable FIR filter with the 8-bit coefficients chosen as the parameters is 56.6% smaller and 28% faster than the generic FIR filter solution. Adapting the FPGA to a new coefficient value only requires 166 μs, while the vendor tools would require 35 s to generate a new specialized circuits and switch to the new circuit. The TLUT tool flow also achieves good results for key-based encryption algorithms. A DCS implementation of TripleDES is 28.7% smaller. Similarly, a DCS implementation of RC6 is 72.7% smaller [7].

B. DCS overhead

Whether or not a DCS implementation is actually beneficial depends on the trade-off between the benefits and the overhead DCS introduces. A DCS implementation can reduce the size and/or increase the speed of the design, but will also introduce an area and a time overhead.

The area overhead is the FPGA-area required to evaluate the tuning functions of the PC and the area needed for internal access to the FPGA configuration memory. Thus, in order to have a net area reduction, the area savings achieved by the DCS implementation should at least offset this area overhead. The evaluation of the tuning functions can be done by any processor. The work of Abouelella [8] presents an overview of the different possibilities. On Virtex-5 FPGAs there are three possibilities: using the PowerPC, a MicroBlaze or a custom processor (Table I).

TABLE I. THE RESOURCE OVERHEAD OF DIFFERENT EVALUATION PLATFORMS ON THE VIRTEX 5 FPGA

Eval. platform	LUTs	BRAM	Clk (Mhz)	Clk/ Boolean Op.
PowerPC	-	-	400	1.04
μBlaze	1532	0	100	1.39
Custom proc.	355	1	295	1

The time overhead (Equation 1) is determined by both the time required for one single specialisation (T_{spec}^{single}) and the number of times a parameter changes value ($\#InputChanges$). Each parameter change requires a specialisation phase, during which the design has to be stalled. Therefore, when designing a DCS implementation, it is important to only select parameters that change infrequently.

$$T_{overh.} = T_{spec}^{single} \cdot \#InputChanges \tag{1}$$

The time needed for one specialisation (T_{spec}^{single}) can be split up in two parts, the evaluation time ($T_{eval.}$) and the reconfiguration time ($T_{reconf.}$). The first is the time needed to evaluate the tuning functions, the second the time for actually reconfiguring the FPGA.

The evaluation time ($T_{eval.}$) is the number of Boolean operations in the tuning functions multiplied by the time to evaluate one Boolean operation on the evaluation platform (See the last column of Table I).

The reconfiguration time ($T_{reconf.}$) depends on the chosen method for reconfiguration and the targeted FPGA. In this paper, a Virtex 5 FPGA is assumed to be the target, but the results can easily be extended to other FPGA architectures. A second assumption is that the Xilinx provided HWICAP port is used to get access to the configuration memory of the FPGA. There are other ways to access this memory, but they fall outside of the scope of this paper. When using the HWICAP for reconfiguration, the FPGA can only be reconfigured frame-by-frame. The LUTs are organised in slice-columns of 20 slices, each slice contains 4 LUTs. All 80 LUT truth tables in a column are spread out over the same 4 frames in such a way that all 4 of them need to be sent, even if only one LUT is reconfigured. The time overhead for reconfiguration is then the number of slice columns with at least one TLUT multiplied by the time needed to send 5 frames over the HWICAP[1].

C. DCS-RTL Profiler

To identify opportunities for DCS, the designer is required to have insight in both the application and Dynamic Circuit specialisation. These are trade-offs a designer is not confronted with in a typical design cycle, as the frequency of signal changes is generally of only secondary importance and estimating the gains from DCS is difficult without actually implementing the DCS design. In RTL design the DCS-RTL profiler, presented in Davidson et al. [3], helps the designer answer this question.

This profiler uses the functional density to compare designs. The functional density (FD) [9], Equation 2, takes both the area (A) and the total execution time (T) into account, allowing the profiler to consider the trade-off between DCS benefits and overhead.

$$FD = \frac{N}{A \cdot T} \tag{2}$$

As it is infeasible to estimate the FD for all signals in the design, this profiler first determines which signals in the RTL-design are good parameter candidates, based on their dynamic behavior. Next, it estimates the functional density (FD) gain of each parameter candidate. To do this, it needs the RTL description of the design and its dynamic behavior. The dynamic behavior data is derived from test bench data. More details can be found in [3].

[1]The fifth frame is a padding frame

III. High-level Profiler

The DCS-RTL profiler offers a solution when an RTL description is already available. However, it would also be useful to detect the opportunities for DCS earlier in the design process, from the high-level description of the design. Converting a high-level description to an RTL implementation involves many design decisions that can restrict the opportunities for DCS. The choices made when designing the RTL implementation significantly impact the potential gains from DCS. For example, a sequential AES implementation that includes a lot of hardware reuse would see almost no benefit from making the key a parameter. A parallel implementation of the same algorithm saves up to 26,7% area by making the key a parameter [10]. In addition, the current digital design trend is moving towards high-level languages, with the help of automated HL-to-RTL tools [11], [12], [13], [14]. Predicting the gains and overhead of DCS from higher-level design descriptions has three major benefits. First, the designer knows earlier in the design process whether a DCS implementation is interesting or not. Secondly, if DCS would be beneficial, this can be taken into account when converting the high-level description to the RTL. Thirdly, the higher abstraction level and the analytical nature of the HL profiler results in very low run-times for the profiler (order of seconds). It avoids the time consuming RTL-to-bitstream steps of the FPGA flow. This allows the profiling itself to happen quickly compared to the RTL-DCS profiler, which can require up to hours for some designs [3]. The HL Profiler presented here takes a high-level description of the application and provides a functional density gain estimate for each parameter candidate. C/C++ was the language of choice in this work as it is supported by a large number of HL-to-RTL tools. However, the principles presented here can be extend to other high-level languages, such as System Verilog, SystemC or Bluespec.

There are a number of ways to translate C/C++-code to RTL code. In most cases, the application is split up in a control path and a data path. Our focus will be on the data path, as the control path contains few opportunities for DCS in most applications. Some signals in the control flow could be good parameters, e.g. if they control the mode of a multi-mode circuit, however, finding these signals requires an analysis on a different scope and is not part of this paper. Multi-mode implementations using the TLUT tool flow are discussed in more detail in [15]. In [10], DCS implementations of different application types are explored. From this work, it is clear that highly parallel RTL implementations benefit more from DCS and that DCS is more interesting for computationally intensive operations, which are generally located in the data path of applications. In addition, our experience with the DCS-RTL Dynamic Profiler informs us that the more parallel an implementation, the larger the proportional area reduction.

A data path generated by the C-to-RTL tools consist of a chain of basic operations [11], [12]. They can be expressed as a hierarchical Data-Flow Graph (DFG), containing three types of nodes: I/O-nodes, data access nodes and arithmetic nodes. It is these arithmetic nodes that are most interesting for DCS. How exactly they are affected by the introduction of DCS is discussed in detail in the next section.

To be clear, the aim of the HL profiler is to give the designer insight into the opportunities for DCS. This is an addition to the normal HLS exploration. The HLS tools give results for several possible implementations, e.g. sequential or parallel, using or not using DSPs, ... The HL profiler can then be used to estimate what the benefits and overhead would be of applying DCS to each of those implementations.

A. *Profiling methodology*

The goal of our profiling methodology is to find out if the application could benefit from DCS. As discussed in II-C, this requires us to consider the trade-off between the DCS gains and overhead. This can be done by using the functional density as metric, as in the RTL-DCS profiler. However, on the higher abstraction level, we have less information on the actual hardware the design is implemented on. Therefore, the metric that will be used is the functional density gain ($FD^{\%}_{gain}$). It is the relative functional density improvement, caused by using DCS. $FD^{\%}_{gain}$ is shown in Equation 3. A_{DCS} is the size of the DCS implementation and $T_{overh.}$ the total time overhead, as in Equation 1. Both A_{DCS} and $T_{overh.}$ are determined by the chosen parameter(s) and have to be estimated with the data available at the higher abstraction level.

$$\begin{aligned} FD^{\%}_{gain} &= \frac{FD_{DCS} - FD_{orig.}}{FD_{orig.}} \\ &= \frac{A_{orig.}}{A_{DCS}} \cdot \frac{T_{orig.}}{T_{DCS}} - 1 \\ &= \frac{A_{orig.}}{A_{DCS}} \cdot \frac{T_{orig.}}{T_{orig.} + T_{overh.}} - 1 \end{aligned} \tag{3}$$

In Equation 3, it is assumed that the DCS implementation will have the same clock as the original implementation. In reality, the area reduction in DCS sometimes leads to improved timing. However, this depends on the actual placement and routing choices, information that is not available to us at this abstraction level. So this means we will possibly underestimate the benefits of DCS slightly, which is not a problem at this stage.

Our profiling methodology works in two stages. First, the dynamic data of signals is collected, to determine which signals would be good parameter candidates. A good parameter candidate is one that introduces a low overhead ($T_{overh.}$). As described in II-B, $T_{overh.}$ depends on both the time needed to reconfigure the application once (T^{single}_{spec}) and the number of times the parameter changes. In the second step, each parameter candidate is considered in turn. For each parameter candidate, the data path is analysed to estimate its impact on each arithmetic node in the DFG. This leads to an A_{DCS} and T^{single}_{spec} for the complete data path, which are combined with the dynamic data from step one to calculate the functional density gain ($FD^{\%}_{gain}$). Both stages are described below in pseudo-code.

```
Analyse the DFG:
Step 1: Dynamic data each input node (Instrumentation)
Step 2:
for Each input node do
    Determine effect on arithmetic nodes (Propagation)
    Analyse each arithmetic node ($T_{eval.}$, $T_{reconf.}$, $A_{DCS}$)
    Estimate $FD^{\%}_{gain}$
end for
```

B. Practical Profiler implementation

The HL Profiler estimates the functional density gain ($FD^{\%}_{gain}$) of adopting DCS, based on the C/C++ description of the design. It requires three pieces of information: the dynamic behavior, the estimated LUT-area (A_{DCS}) and the estimated overhead ($T_{overh.}$). How each of these is gathered is discussed in detail below.

1) Dynamic data: First, the dynamic data has to be gathered. To do this, a hierarchical Data-Flow Graph (DFG) of the C-description of the application is generated. This DFG contains three types of nodes: I/O-nodes, data access nodes and arithmetic nodes. To collect the dynamic data on all parameter candidates, all I/O nodes of the DFG are instrumented in the original C-code, and the application is executed with realistic input data. This information will help estimate $T_{overh.}$, together with T^{single}_{spec}, which is estimated in the next section. We know that parameter candidates with fewer value changes are more interesting. Therefore, the average time between parameter changes is used to rank the candidates. The parameter candidate that changes least frequently is considered first. To collect this data, only the number of times an input changes needs to be registered. The exact moments of the value changes are not important.

A possibility for future research in this step is to extend it to also consider multiple input nodes as parameters at the same time. This introduces two problems: First, the total number of parameter candidates will need to be restricted in some way, otherwise it can grow too large. Secondly, considering multiple input nodes at the same time requires us to also collect the exact timing of the value changes. Without this information, the dynamic behavior of the combined input nodes can not be determined.

2) Data path analysis (A_{DCS} & $T_{overh.}$): The DFG is also used to determine which arithmetic nodes are affected by the introduction of DCS. The DFG is analysed for every parameter candidate and this is done in different steps. The first step, the *parameter propagation*, starts by marking the current candidate, or I/O-node, as a parameter. Then, this is propagated through the complete DFG. The output of each arithmetic node that has only parameter inputs is also marked as a parameter, until no new parameter markings can be made.

In the second step, we determine the size and DCS overhead of each arithmetic and output node in the DFG. The size of a node can be reduced if it can be implemented through DCS. However, then it will add to the complexity of the tuning functions (See Section II-A), thus increasing the time overhead. Each node is analysed based on its inputs. There are three possibilities: nodes either have (i) no parameter inputs, (ii) only parameter inputs or (iii) some parameter inputs.

Nodes that fall under case (i) are unchanged and nodes with only parameter inputs (ii) are completely removed from the design. The removed nodes will not be implemented in hardware, so they will not add TLUTs to the design, but their functionality is captured in the tuning functions. To know how many boolean operations a node adds to the tuning functions once it is removed, it needs to be analysed. This was done below for the addition and the multiplication node.

The total number of boolean operations added by a node is B_{node}. The most reliable metric for this is the number of LUTs the original implementation required. To get B_{node}, the number of original LUTs is multiplied by B_{avg} (Equation 4). B_{avg} is the average number of Boolean operations for each original LUT in the node and was determined for both the addition and the multiplication node (Table II). For the addition node, the number of original LUTs is the number of output bits, or the largest input size plus one. Only one B_{avg} is needed to be accurate for all adder sizes. The multiplier is more complicated. In a multiplication node, the number of original LUTs is the product of the input bit sizes. To have an accurate enough estimate, a different B_{avg} value has to be used depending on the smallest sized input. Table II shows both the most and least accurate B_{avg} value. The total number of Boolean operations a removed node adds to the tuning functions (B^{node}) can be calculated from this data, using Equation 4.

TABLE II. BOOLEAN OPERATIONS.

Node	Avg Boolean Ops /Original LUT (B_{avg})	STDEV
Adder	145	5.9
Multiplier	50 - 64	0 - 5.6

$$B_{node} = B_{avg} \cdot \#\text{original LUTs} \quad (4)$$

If an output node has a parameter input, then it is implemented by N TLUTs, with N the bit width of the output. The TLUTs are used to store the parameter value. The logic function of the removed nodes before the output node are now executed through the evaluation of the Boolean functions. The cost of this was already accounted for when these nodes were removed, the only cost associated with the output node is the addition of the TLUTs.

If only some inputs of the arithmetic node are parameters (case iii), then the analysis depends on the arithmetic function of the node. For example, adders will not benefit from having one parameter input. Indeed, an absolute lower bound for the size of any node is the number of output bits it drives. The number of output bits of an adder is the largest of both input bit sizes plus one. Current FPGA architecture(s) and CAD tools always succeed in implementing adders in this lower bound, therefore DCS can not further improve the area of the adder itself. However, DCS does replace the LUTs of the adder with TLUTs and will remove the parameter input from the design. This will reduce the routing requires to implement the adder. However, since this effect is very hard to account for at the higher abstraction level we will not consider it. Similar to the output node with parameter inputs, the tuning functions of the TLUTs in the adder node will contain the arithmetic functions of the removed nodes, if there were any.

An arithmetic node that does show benefits when only one input is a parameter is the multiplication node. In that case, the FPGA area required for this node reduces significantly. However, a more detailed discussion is necessary to determine exactly how large this decrease is, and to allow an estimate of the overhead this introduces.

Multiplication Node: In general, the TLUT tool flow is able to convert any VHDL-description to a DCS implementation. Of course, this does not mean all applications will show gains. Some applications lend themselves better to DCS. Even implementations of the same application show very different gains with DCS [10]. For a DCS multiplier, a hierarchical design has the best results. E.g. a DCS-implemented 24-bit x 24-bit multiplier has a 54.3% area reduction with a hierarchical design. Without it, the area reduction is only 18.9%. This holds both for both smaller and larger multipliers.

In a hierarchical multiplier implementation, the multiplier is split up in smaller sub-multipliers. Each of the smaller sub-multiplier results is shifted and added to produce the final multiplication result. Our research has shown that there is an optimal sub-multiplier size, depending on the number of inputs the LUTs of the target FPGA have. To focus the discussion, assume that the top level has a P-bit parameter input and a N-bit normal input and that we are targeting a 6-input LUT FPGA. Almost all modern FPGAs use 6-input LUTs. Under those conditions, the optimal sub-multiplier size is a 6-bit by P-bit multiplier. The 6-bit sub-multipliers can always be implemented in only $6 + P$ TLUTs, regardless of parameter size. This is exactly the number of output bits of the 6-bit by P-bit multiplication, the absolute lower bound for implementing a node. In a 6-input LUT FPGA, this is the largest size for which this is possible. A similar hierarchical multiplier can also be built from 5-bit by P-bit, or smaller, sub-multipliers. However, they require more (T)LUTs for the complete multiplication and a larger adder structure. In our datapath analysis, we assume a hierarchical multiplier with 6-bit x P-bit sub-multipliers. If the size of the regular input is not divisible by 6, an extra multiplier of the appropriate size (5, 4, 3, 2 or 1 bit) is added, for processing the remainder. An example of the complete hierarchical 24-bit by 24-bit multiplier is shown in Figure 1.

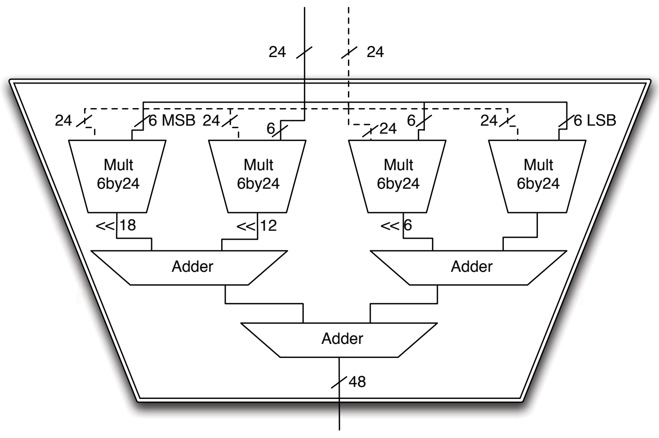

Fig. 1. Hierarchical 24-bit x 24-bit multiplier, dotted line for the parameter input

Aside from the better LUT-area gains, the hierarchical structure of the multipliers is very useful. It allows estimates on all aspects of the top-level based on the sub-multiplier data. The first property we need is the area of the hierarchical multiplier. The bit-shifts can be implemented by the interconnections and take no FPGA area. This leaves the sub-multipliers and the adder-tree. The number of 6-input sub-multipliers is $M = \lceil N/6 \rceil$, with N still the regular input bit-size. Each of the 6-bit sub-multipliers requires $6 + P$ TLUTs. The MSB multiplier is actually an R sized sub-multiplier, with $R = N - ((M - 1) \cdot 6)$. Its size is $R + P$. The total size of all sub-multipliers in the design is shown in Equation 5.

$$Size_{sub-mults} = M \cdot (6 + P) - (6 - R) \quad (5)$$

The results of the multiplication are shifted by multiples of 6 and then added to get the final result. The addition is done through an adder tree. The first level adds the shifted results of the sub-multipliers two by two, every following level does the same with the results of the previous adder level, until only one value is left. This is also shown in Figure 1. The size of these adders is also a factor in the total multiplier size. For sake of clarity, the description below assumes N is divisible by 6 and the number of sub-multipliers is even. The first level of adders adds two $6 + P$-bit operands together, but the MSB operand of each addition is shifted over 6 bit positions. This means the 6 LSBs of each addition are not actually added, making the size of each level one adder $6 + P + 1$ LUTs. However, each level one adder does output $12 + P + 1$ bits to the second stage adders, the 6 LSBs are just copied. In the second stage, the MSB operand is shifted 12 bit positions. The adder size is $12 + P + 2$. This pattern holds for the later stages. This is an idealised description, in reality, both an uneven number of multipliers and the smaller multiplier for the remainder have to be taken into account. An uneven number of sub-multipliers usually leads to intermediate results skipping a stage, resulting in smaller adders. The exact number of adders of each size is a function of M, the number of sub-multipliers $(F(n, M))$. The remainder sub-multiplier is corrected by subtracting $(6 - R)$ for each adder that handles the output of the MSB sub-multiplier $(CF = C(M) \cdot (6 - R))$. The number of stages is $S = \lfloor LOG(M, 2) \rfloor$. The total size of the adder is then given by Equation 6.

$$Size_{adders} = \sum_{n=1}^{S} F(n, M) \cdot (6 + 2^{n-1}(P + 1)) - CF \quad (6)$$

Combining Equation 5 and Equation 6 yields the complete size of the hierarchical multiplier.

Not only the area of the multiplier is important, but also the time overhead specializing introduces. This will determine the impact of the multiplier on the T_{spec}^{single} of the DCS implementation. The single specialisation overhead is split up in two parts: the evaluation overhead and the reconfiguration overhead. For both estimates, the hierarchical structure of the multipliers is exploited. Both depend in some way on the number of TLUTs, the only LUTs that are reconfigured.

$$B^{mult} = ((M - 1) \cdot B_6 + B_R) \cdot P \quad (7)$$

The evaluation overhead, Equation 7, is the time needed to calculate the new bit values of the FPGA circuit. It is evaluating the tuning functions that constitutes this overhead. Analysis of the 6-bit sub-multiplier shows that each additional parameter bit increases the number of Boolean operations by 1903. For the MSB sub-multiplier, the added Boolean functions depends on its size (R), see Table III. The actual evaluation overhead is also dependent on the platform that will evaluate them. The next section will combine this data with the data on the removed nodes to calculate the T^{eval} of the complete data path.

The reconfiguration time overhead is a direct function of the number of TLUTs. For each sub-multiplier the number of

TABLE III. BOOLEAN OPERATIONS FOR 1-TO-6 BIT SUB-MULTIPLIER

Sub-mult size	Boolean Ops.	STDEV
B_1	64	0.47%
B_2	123	4.95%
B_3	230	1.64%
B_4	467	0.43%
B_5	937	1.38%
B_6	1903	2.67%

TLUTs is directly proportional to the dimensions of the inputs, it is the sum of parameter (P) and regular input bits (6). The sub-multipliers are the only part of the hierarchical multiplier that contain TLUTs, the other parts are all implemented in normal LUTs. The reconfiguration overhead is dependent on the total number of TLUTs in the datapath and will be discussed in detail in the next section. Since all of the sub-multiplier LUTs are TLUTs and no other parts of the multiplier contain TLUTs, the number of TLUTs is exactly the size of the sub-multipliers (Equation 8).

$$TLUTs^{mult} = M \cdot (6 + P) - (6 - R) \tag{8}$$

This analysis gives us the size ($Size_{sub_mults} + Size_{adders}$), the size of the Boolean functions (B^{mult}) and the number of TLUTs ($TLUTS^{mult}$) a multiplier with only one parameter input introduces in the DCS implementation. This information gives us all the information necessary to include these multipliers in the functional density gain estimate. An addition node does not require such an extensive analysis, as it does not benefit from having only one parameter input. If the datapath also contains nodes with a different arithmetic function, they would require a similar analysis.

3) $FD^{\%}_{gain}$ estimate: To estimate the functional density impact of a parameter candidate, both the area savings and the time overhead it introduces have to be estimated. These estimates depend on both the dynamic data gathered through instrumentation and on the datapath analysis.

The total overhead introduced by DCS (Equation 1) is defined by the number of times the parameter candidate changes and the overhead of a single specialisation. The first is collected through the dynamic data, by instrumenting the I/O nodes in the datapath.

$$T_{overh} = T^{single}_{spec} \cdot \#InputChanges$$

The estimate of T^{single}_{spec} is the sum of T_{eval} and T_{reconf}. T_{eval} is determined by the removed nodes and the multiplier nodes with some, but not only, parameter inputs. The removed nodes are replaced by tuning functions, just as some of the logic in the multiplier nodes. The total number of Boolean operations in the tuning functions is the sum of B^{node} for all removed nodes and B^{mult} for each multiplier with only one parameter input. The time required for each Boolean operation (T_{bool}) depends on the targeted evaluation platform. As table I shows, the embedded PowerPC on some Virtex5 FPGAs, requires 1.04 clock cycles.

$$T_{eval} = (\sum B^{node} + \sum B^{mult}) \cdot T_{bool} \tag{9}$$

T_{reconf} is dependent on the total number of TLUTs in the datapath. TLUTs are only added to the design by nodes where the parameter propagation terminates, i.e. either an output node or an arithmetic node with at least one regular input. In the first case, the output node will require P TLUTs to be implemented, with P the number of parameter bits. In the second case, assuming a general arithmetic node, the node size will not change. However, in the worst case, all LUTs of the arithmetic node are now TLUTs. An exception is made for multiplication nodes who benefit significantly from having a parameter input. In that case, the number of TLUTs, as described in Equation 8 is used.

Once the total number of TLUTs is known the reconfiguration time can be estimated. To get the reconfiguration overhead, we need to estimate the number of FPGA frames that are reconfigured. To get the number of frames, the structure of the FPGA configuration memory has to be taken into account too. A Virtex-5 FPGA can only be reconfigured tile by tile. Each LUT-tile contains 20 slices and each slice 4 LUTs. To reconfigure one LUT in one slice, the data for all 20 slices needs to be sent[2]. So, to estimate the total reconfiguration overhead, we need to determine how many tiles will contain at least one TLUT. Several assumptions are made: (i) The complete design will be placed in a rectangular slice-area, covering C tiles with $C = \lceil\sqrt{LUTs}) \cdot 1.7\rceil \cdot \lceil\frac{\sqrt{LUTs}}{80 \cdot 1.7}\rceil$. (ii) The TLUTs are spread out uniformly over this area and are placed in groups of 5. Under those assumptions, the number of tiles containing a TLUT ($E[\#Tiles]$) is given by Equation 10. The correction factor of 1.7 for C and the decision to cluster the TLUT groups of 5 were shown to yield the most accurate estimates in experiments.

$$\begin{aligned} E[\#Tiles] &= C \cdot P(tlut \in Tile) \\ &= C \cdot (1 - P(tlut \notin Tile)) \\ &= C \cdot \left(1 - \prod_{n=0}^{tlut-1} \left(\frac{(C \cdot 80 - n) - 80}{C \cdot 80 - n}\right)\right) \end{aligned} \tag{10}$$

The reconfiguration time is then the multiplication of the number of columns containing at least one TLUT ($E[\#Tiles]$) and the time reconfigure one tile (T_{tile}). On a Virtex 5 FPGA, T_{tile} is 40.9 μs.

$$T_{reconf} = T_{tile} \cdot E[\#Tile] \tag{11}$$

Combining Equations 9 and 11 yields the estimate for the single specialisation time (T^{single}_{spec}), Equation 12 and thus, also the total overhead ($T_{overh.}$.

$$\begin{aligned} T^{single}_{spec} &= T_{eval} + T_{reconf} \\ &= T_{tile} \cdot E[\#Tile] + \left(\sum B^{node} + \sum B^{mult}\right) \cdot T_{bool} \end{aligned} \tag{12}$$

The total area savings can be estimated in a similar way, based on the results of the data path analysis. Nodes with no parameter inputs are unchanged. Each node with only parameter inputs is assumed to require no LUT area. This leaves the nodes with at least one parameter input. These were already discussed in the previous paragraph. Except for the output and the multiplier node, they are assumed to require the same area. The combination of all these estimates is the final A_{DCS} estimate.

[2]Four frames and a padding frame.

Now both T_{spec}^{total} and A_{DCS} are known, it is possible to calculate the functional density gain (Equation 13). However, this does requires us to know both the size of the original datapath ($A_{orig.}$) and its execution time ($T_{orig.}$). $A_{orig.}$ can be estimated using the tools of [16]. It also estimates the size of application by analysing the datapath and providing a LUT size estimate for each arithmetic node. It can not take dynamic reconfiguration into account. For example, this tool estimates the size of a multiplier to be the product of its operand sizes. This means the original size of a 24-by-24 bit multiplier is 576 LUTs, a result that matches our own experiments. A DCS implementation of the same multiplier requires only 313 LUTs. $T_{orig.}$ can be difficult to determine. The execution time of the original C-code could be used, but then the designer should first verify that the timing of the hardware implementation is similar to the timing of the C-code. Because hardware implements functions, its timing can differ significantly from the high-level description. Another option is to take a set amount of execution time and then use the first stage of the HL profiler to measure the average parameter behaviour over that time. However, this does require a representative set of input values for that time-period. The set amount of execution time should be long enough to capture a 'typical' behaviour and at least several magnitudes larger than 40.9 μs, the minimal $T_{overhead}$. Another option is to capture the execution time from an RTL implementation of the original data path, using a C-to-RTL tool such as Vivado HLS [1].

$$FD_{gain}^{\%} = \frac{A_{orig.}}{A_{DCS}} \cdot \frac{T_{orig.}}{T_{orig.} + T_{overh.}} - 1$$

The $FD_{gain}^{\%}$ is calculated for each parameter candidate and based on these predictions, the designer can decide if, for a specific parameter candidate, a DCS implementation would be worth to explore further.

IV. Experiments

In order to verify the methodology presented in the previous section, it was implemented and tested for a set of PID designs. This design was chosen because we were able to quickly build an optimized DCS implementation that scales with the dimension of the inputs. The optimized DCS implementation is necessary as a comparison for the HL profiler results. In the future, more extensive testing is necessary, especially on larger designs. A PID controller is a control loop feedback mechanism that incorporates the current error (*P*roportional), the past errors (*I*ntegral) and a prediction of the future error (*D*erivative). This digital PID implementation is discussed in [17]. The calculations it executes are the following:

$$\begin{aligned} Error &= input - current\ possition \\ O1 &= Error + O1 \\ O2 &= Error + \frac{O2}{2} \\ Outp &= F1 \cdot O1 + F2 \cdot O2 + C \cdot Error \end{aligned} \tag{13}$$

This data path contains 5 adders and 3 multipliers. If Y is the dimension of the input, then there are three adder with Y-sized inputs, two with 2*Y sized inputs and three multipliers with Y-sized inputs. The opportunity for DCS in this case are the coefficients ($F1, F2, C$). They are not constant but change very infrequently, making them good parameter candidates. All three of them are considered parameters at the same time.

TABLE IV. HL profiler result for different PID implementations. The errors are listed between brackets.

PID	A_{orig}	HL Prof A_{DCS}	Boolean Ops	TLUTs	Frames
6-bit	150	78 (-27%)	34254 (12%)	36	29 (38%)
12-bit	516	288 (-10%)	137016 (1.8%)	108	65 (-10%)
18-bit	1098	606 (4.6%)	308286 (-0.4%)	216	105 (-31%)
24-bit	1896	1107 (16%)	548064 (-1.4%)	360	149 (27%)
30-bit	2910	1659 (22%)	856350 (-2.1%)	540	193 (-5.8%)

TABLE V. DCS implementation result for different PID implementations.

PID	A_{orig}	DCS implementation A_{DCS}	Boolean Ops	TLUTs	Frames
6-bit	150	108	30474	36	21
12-bit	516	321	134544	108	73
18-bit	1098	579	309546	216	153
24-bit	1896	951	556332	360	117
30-bit	2910	1358	875010	540	205

In our experiment, the target FPGA was a Virtex 5 FPGA with an embedded PowerPC (XC5VFX70T-1FF1136). The PowerPC is used to evaluate the Boolean functions and only LUTs are used to implement the design. The input dimension of the PID design was varied from 6 to 30 bits. Table IV shows the HL profiling estimates for the size of the DCS implementation, the number of Boolean operations, TLUTs and frames. The % error is shown next to these values, these errors are compared to the data on the actual DCS implementation, shown in Table V.

The most accurate estimate of the HL profiler is clearly the number of TLUTs in the design, it always matches the actual number of TLUTs exactly. Sadly, this does not lead to a very accurate frame estimate. This is as expected, this estimate depends on the place and route steps of the FPGA tool flow, something we have almost no information off on this higher abstraction level. Moreover, the placement and routing steps themselves are not always predictable. For example, while the 24-bit design is larger and contains more TLUTs than the 18-bit design, its DCS implementation requires less frames. The number of Boolean operations is predicted quite accurately. The worst case is for the smallest design, with an error of 11.7%. The results for the larger designs are significantly better, fluctuating slightly around a 2% error. The size estimate of the DCS design is not very accurate. The worst case is the 6-bit design, where A_{DCS} is underestimated by 30%. The other cases are more accurate and the larger design sizes tend to be overestimated. An overestimate of A_{DCS}, and thus an underestimate of the benefits from DCS, is more acceptable for an HLS profiler. Clearly, some effect is not captured well enough by our current analysis and should be researched further.

While the estimates in the HL profiler, except for the number of TLUTs, clearly should be improved further, it does allows the designer to have some insight in the opportunities for DCS. At this abstraction level, our methodology is the only one that can provide any information on these opportunities.

V. Conclusion

In this paper we have presented a high-level profiler that predicts the benefits and drawbacks of applying DCS to a design, based on its C/C++-description. These predictions are based on analytical functions and heuristics, allowing them to be calculated very quickly. In addition, because this tool works on a higher abstraction level, it allows a fast design space exploration that includes Dynamic Circuit Specialisation implementations. Based on the experiment discussed in the previous section, it is clear that the estimates in the HL profiler should be improved further to increase the general usefulness of the profiler. Currently, only the estimates for the number of TLUTs and the number of Boolean operations are accurate enough. When improving the estimates, the end goal should be a lower bound prediction for the benefits of DCS. If that is the case, then the designer is sure to make a decision based on the worst-case scenario. Right now, the results do allow the designer insight in the opportunities for DCS. At this abstraction level, our methodology is the only one that can provide any information on these opportunities.

References

[1] T. Feist, "Vivado design suite," 2012.

[2] D. Lim and M. Peattie, *Two flows for partial reconfiguration: Module based or small bit manipulations. Xilinx Application Note 290 (v1.0)*, 2002.

[3] T. Davidson, K. Bruneel, and D. Stroobandt, "Identifying opportunities for dynamic circuit specialization," in *Workshop on Self-Awareness in Reconfigurable Computing Systems, Proceedings*, London, UK, 2012, pp. 18–21.

[4] C. Ababei, "Speeding up FPGA placement via partitioning and multi-threading," *International Journal of Reconfigurable Computing*, 2009.

[5] K. Bruneel, W. Heirman, and D. Stroobandt, "Dynamic data folding with parameterizable fpga configurations," *ACM TRANSACTIONS ON DESIGN AUTOMATION OF ELECTRONIC SYSTEMS*, vol. 16, no. 4, p. 29, 2011.

[6] HES Group, Ghent Univeristy. (2012) TMAP toolflow.

[7] T. Davidson, F. Abouelella, K. Bruneel, and D. Stroobandt, "Dynamic circuit specialisation for key-based encryption algorithms and DNA alignment," *International Journal of Reconfigurable Computing*, 2011.

[8] F. Mostafa Mohamed Ahmed Abouelella, T. Davidson, W. Meeus, K. Bruneel, and D. Stroobandt, "How to efficiently implement dynamic circuit specialization systems," *ACM TRANSACTIONS ON DESIGN AUTOMATION OF ELECTRONIC SYSTEMS*, p. 38, 2013.

[9] A. M. Dehon, *Reconfigurable architectures for general-purpose computing*. Massachusetts Institute of Technology, 1996.

[10] T. Davidson, F. Abouelella, K. Bruneel, and D. Stroobandt, "Dynamic circuit specialisation for key-based encryption algorithms and DNA alignment," *International Journal of Reconfigurable Computing*, 2012.

[11] S. Gupta, R. K. Gupta, N. D. Dutt, and A. Nicolau, *SPARK: a parallelizing approach to the high-level synthesis of digital circuits*. Springer, 2004.

[12] D. Chen, J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "xpilot: A platform-based behavioral synthesis system," *SRC TechCon*, vol. 5, 2005.

[13] N. Kavvadias and K. Masselos, "The hercules high-level synthesis environment," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, 2013, pp. 1–1.

[14] R. Nikhil, "Bluespec system verilog: efficient, correct rtl from high level specifications," in *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*. IEEE, 2004, pp. 69–70.

[15] B. Al Farisi, K. Bruneel, J. M. P. Cardoso, and D. Stroobandt, "An automatic tool flow for the combined implementation of multi-mode circuits," in *Proceedings - Design, Automation, and Test in Europe Conference and Exhibition*, Grenoble, France, 2013, pp. 821–826.

[16] X. Niu, T. C. Chau, Q. Jin, W. Luk, and Q. Liu, "Automating resource optimisation in reconfigurable design (abstract only)," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '13, 2013, pp. 275–275.

[17] R. Blomme, D. Brokken, and J. Van Campenhout, "Eindverslag robotica robotsturing iwonl conventie nr. 4930 derde biennale 1/9-30/11/87," Gent, Tech. Rep., 1987.