Mapping Loop Structures onto Parametrized Hardware Pipelines

Adrien Le Masle and Wayne Luk, *Fellow, IEEE*

***Abstract*— This paper shows how a general form of algorithms consisting of a loop with loop dependencies carried from one iteration to the next can automatically be mapped to a parametric hardware design with pipelining and replication features. A technology-independent parametric model of the proposed design is developed to capture the variations of area and throughput with the number of pipeline stages and replications. Our model allows rapid optimization of design parameters by a few pre-synthesis operations. We present an optimization method based on the model. Our method is evaluated using three different applications implemented on a Xilinx Spartan 6 XC6SLX45T FPGA: a carry-save adder-based Montgomery multiplier, a modular exponentiation module, and an integer square root module. Our model facilitates design space exploration; it can quickly predict the area taken by our designs with less than 5% of error, and their maximum frequencies and throughputs with less than 22% of error. Our optimization method is up to 96 times faster than a full search through the design space.**

***Index Terms*— Design space exploration, field-programmable gate array (FPGA), hardware mapping, loop-carried dependencies, resource estimation.**

I. Introduction

FIELD-programmable gate arrays (FPGAs) are quickly increasing in capability and in size, and it becomes a growing challenge to fully cover the design space available for a given budget. This introduces the need for parametric designs capable of covering a large design space, especially in terms of the speed–area tradeoff. Recent papers have explored the benefit of parametric designs in different applications. In [1], a parametric approach is developed for software-defined radio modules. In [2], we present parametric Montgomery multiplier, Montgomery exponentiator, and Miller–Rabin primality tester designs. These three functions are the core of many public-key crypto-systems.

The main advantages of parametric designs are their scalability and their reusability. They can be reused as IP cores in many projects with different speed–area tradeoffs. However, finding the optimal values for the parameters to meet a given design goal can be difficult and time-consuming. An exhaustive approach implementing designs for all the values of the parameters in a given search interval cannot be achieved in most projects where the time-to-market is an issue. Therefore, we need a model of the design that facilitates estimating its parameters under speed and area constraints. Such area and speed estimation models are often application-dependent [1], [3] or technology-dependent [4], [5].

In this paper, we present a general parametric hardware design method using pipelining and replication. We show how algorithms in a particular form, namely, those containing one loop with loop dependencies carried from one iteration to the next, are automatically mapped to appropriate hardware designs. Then we develop a model of the area and throughput of the design. Our main contributions include:

1) a coarse-grained method for mapping an algorithm containing one loop with loop dependencies carried from one iteration to the next to a parametric design using pipelining and replication;
2) a model of our design that allows predicting the variations of its throughput and its area with the number of pipeline stages and replications;
3) a general optimization process integrating our model;
4) a practical analysis of the accuracy and prediction capabilities of our optimization method on three applications implemented on a Xilinx Spartan 6 XC6SLX45T FPGA.

Our model is technology-independent. It allows rapid optimization of the design by only running a few pre-synthesis operations. It facilitates design space exploration; it can quickly predict the area taken by the three evaluated designs with less than 5% of error, and their maximum frequencies and throughputs with less than 22% of error. As a matter of fact, our optimization method is up to 96 times faster than a full search through the design space.

The rest of this paper is organized as follows. Section III presents the general algorithm and its hardware mapping. Section IV develops an area, frequency, and throughput model of our hardware mapping. Section V shows how this model is integrated into a throughput optimization process. Section VI evaluates the accuracy and prediction capabilities of our optimization method and shows that it greatly speeds up design space exploration. Finally, Section VII concludes this paper.

II. Related Work

A. Resource and Performance Estimations

Previous work on resource and performance estimations for reconfigurable devices has mainly been focused on high-level

Manuscript received December 20, 2011; revised January 22, 2013; accepted February 11, 2013. Date of publication March 28, 2013; date of current version February 20, 2014. This work was supported in part by BlueRISC, Xilinx, U.K. EPSRC, the European Union Seventh Framework Programme under Grant Agreement 248976, Grant Agreement 257906, Grant Agreement 287804, and Grant Agreement 318521, and the HiPEAC NoE.

The authors are with the Department of Computing, Imperial College London, London SW7 2BZ, U.K. (e-mail: al1108@doc.ic.ac.uk; wl@doc.ic.ac.uk).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2013.2251430

register-transfer level (RTL) estimation. The common method used for resource estimation consists of identifying the basic operations in the algorithm, which can be represented as a data flow graph [4] or an RTL netlist [5]. A library of operators characterizing the device is then used to estimate the resources needed by the algorithm. This process usually requires approximations to fit the parameters of identified operations (bitwidth, number of inputs, etc.) to operators in the library [6]. Such estimation tools are readily available in commercial software, such as Xilinx system generator [7] and PlanAhead [5]. The RTL resource estimation tool used by Xilinx is claimed to be 60 times faster than synthesis [5], which allows fast design space exploration. Similar methods can be used to estimate the performance of the design by using a device-dependent library of delays for the estimated resources.

These methods are general and inherently fine-grained. A coarse-grained approach focused on energy-consumption has been proposed [1], [8], which targets runtime reconfigurable software-defined radio applications. This approach needs a priori information about the implementation, usually obtained by a few synthesis operations.

B. Algorithm Mapping onto Reconfigurable Hardware

The literature on algorithm mapping onto reconfigurable hardware and associated design space exploration is vast. For instance, in [9], a mixed compiler/synthesis tools approach is presented. Compiler optimizations are performed on the algorithm before translation to RTL. The design space is then searched using RTL estimations as described earlier.

The mapping of an unrolled loop onto pipelines has also been extensively studied [10]–[12]. The reconfigurable dataflow approach [12], in particular, maps loops with loop-carried dependencies onto efficient pipelines. This method is more fine-grained and general than our approach, since it supports loop-carried dependencies of various sizes and loop indices determined at runtime. However, resource and performance models are not covered for the mapped implementation, which makes design space exploration difficult.

Unlike other more fine-grained loop pipelining and resource estimation methods, the method presented in this paper is restrained to a particular application domain and uses a coarse-grained model of the corresponding hardware mapping. As a matter of fact, our base module is a coarse-grained cell, which already consists of many fine-grained operators. Similar to other methods [1], [8], our approach is particularly useful for quick optimization of design parameters through design space exploration. However, we focus on algorithms that cannot be parallelized due to loop-carried dependencies, while other methods focus on tradeoffs related to parallelization [1], [8]. The restrained application domain of our model makes the resource and performance estimations as well as the design space exploration more accurate than general estimation methods.

Our coarse-grained estimations and mapping are performed before synthesis. Other fine-grained pipelining techniques, such as post-placement C-slow retiming [13] can be performed after synthesis if the throughput of the design needs to be increased further. However, such techniques can have non-negligible overheads that should be taken into account.

Algorithm 1 CSA Montgomery

Input: $A = (a_{n-1}...a_0)_2$, $B = (b_{n-1}...b_0)_2$, $M = (m_{n-1}...m_0)_2$
Output: $P = A.B.2^{-n} \bmod M$

```
1 S = 0, C = 0, D = B + M
2 for i = 0 to n − 1 do
3     I = select(s_0, c_0, a_i, B, M, D)
4     S, C = S + C + I
5     S = S div 2, C = C div 2
6 end
7 P = S + C
8 if P ≥ M then P = P − M
```

III. Hardware Design With Pipelining and Replication

We develop a hardware design using pipelining and replication for a general form of algorithms, which can be found at the core of many applications.

A. Main Idea

This paper focuses on a class of algorithms with loop dependencies carried from one iteration to the next. Common applications falling into this category are bitwise algorithms for which the result is updated at each iteration based on the corresponding bit of one or several inputs. In cryptography, this is for instance the case of the Montgomery multiplication algorithm [14], the square-multiply exponentiation algorithm [2], and the Lucas primality test [15], [16]. Some important arithmetic algorithms, such as the integer square root [17] and the restoring division algorithms also follow this pattern. Algorithms performing a reduce operation also exhibit such loop dependencies. This is the case of several algebraic algorithms, such as matrix-vector and matrix multiplications, for which products of row and column elements are iteratively accumulated. Finally, loop dependencies carried from one iteration to the next are also common in approximation algorithms for which a solution is refined at each iteration based on the result from the previous one. The Newton–Raphson method is one of the best known algorithms of this type.

Usually algorithms with loop-carried dependencies cannot be easily parallelized. However, several techniques can be used for efficient hardware implementation. As an example, let us consider carry-save adder (CSA)-based Montgomery multiplication shown in Algorithm 1. After initialization, the algorithm iterates on the bits of the input A. At each iteration, the new sum S and carry C are computed using the current sum, the current carry and a number I. The value of I depends on the LSB of the current sum, the current carry and the input B, as well as bit i of A. S and C are then shifted right. After the loop terminates, the result is converted from carry-save to normal representation and reduced modulo M.

For computing a 32-bit ($n = 32$) Montgomery multiplication, a simple hardware architecture for the loop would

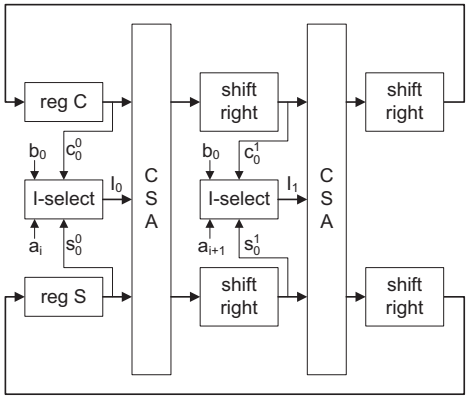

Fig. 1. Focus on a Montgomery block for $r = 2$.

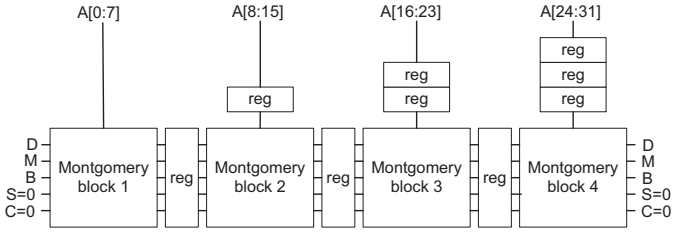

Fig. 2. Structure of a 32-bit pipelined Montgomery multiplier with four pipeline stages.

reuse the same processing logic (called Montgomery cell) 32 times, storing the intermediate values of S and C in registers. This architecture is compact but slow as the latency of one multiplication is 32 clock cycles. If we are willing to trade area for performance, the Montgomery cell can be replicated. For example if we replicate the Montgomery cell once, we obtain $r = 2$ copies of the cell in series in a Montgomery block, as shown in Fig. 1. Hence, the latency of one multiplication is now $32/2 = 16$ clock cycles. The area is slightly less than doubled as some control logic can be shared. However by replicating the cell, we increase the delay through the block and hence the critical path delay. In practice, replication is only useful if the minimum period that we want to achieve is higher than the critical path delay. Now let us assume that we want to perform several multiplications. In our replicated design, we would have to wait for the first multiplication to terminate before starting a new one, leading to a throughput of one multiplication every 16 clock cycles. To increase the throughput of the design, we can pipeline the Montgomery block as shown in Fig. 2. As opposed to replication, pipelining buffers the output of each duplicated block. Here, the entire block is duplicated, and therefore doubling the number of pipeline stages doubles the logic area. For $p = 4$ pipeline stages, each block is assigned $32/4 = 8$ iterations. Each block contains $r = 2$ cells. Hence, it takes $8/2 = 4$ clock cycles for each block to compute its part of the iterations. The latency of the architecture is still 16 clock cycles but a new multiplication can now be started every four clock cycles, leading to a throughput of one multiplication every four clock cycles. The following sections formalize these ideas.

Algorithm 2 General Bit by Bit Processing Algorithm

Input:
$A_0 = \sum_{i=0}^{n-1} a_{(0,i)}2^i, \ldots,$
$A_{k-1} = \sum_{i=0}^{n-1} a_{(k-1,i)}2^i,$
$B_0 = \sum_{i=0}^{\beta_0-1} b_{(0,i)}2^i, \ldots,$
$B_{l-1} = \sum_{i=0}^{\beta_{l-1}-1} b_{(l-1,i)}2^i$
Output:
$R_0 = \sum_{i=0}^{\rho_0-1} r_{(0,i)}2^i, \ldots,$
$R_{m-1} = \sum_{i=0}^{\rho_{m-1}-1} r_{(m-1,i)}2^i$

1 $(R_0, \ldots, R_{m-1}) = pre(A_0, \ldots, A_{k-1}, B_0, \ldots, B_{l-1})$
2 **for** $i = 0$ **to** $n - 1$ **do**
3 $(R_0, \ldots, R_{m-1}) = main(a_{(0,i)}, \ldots, a_{(k-1,i)}, B_0, \ldots, B_{l-1}, R_0, \ldots, R_{m-1})$
4 **end**
5 $(R_0, \ldots, R_{m-1}) = post(A_0, \ldots, A_{k-1}, B_0, \ldots, B_{l-1}, R_0, \ldots, R_{m-1})$

B. General Algorithm

In this analysis, we consider algorithms that can be described in the general form presented in Algorithm 2.

This algorithm has an optional pre-computation stage (line 1). This stage can consist of initialization of variables, initial shifts, etc. Then, the algorithm iterates on the number of bits of the A inputs (A_0 to A_{k-1}). At each iteration of the loop, the result variables $R_0, \ldots, R_{m-1}$ are updated through the main() function (line 3). At iteration i, this function takes the previous values of $R_0, \ldots, R_{m-1}$, the entire B inputs (B_0 to B_{l-1}), and the i-th bits of the A inputs. A possible post-computation stage terminates the algorithm (line 5).

In practice, every algorithm with one loop that has loop dependencies carried from one iteration to the next, and for which some inputs are processed bit-by-bit, can be represented in this general form. Table I shows how we can map the Montgomery multiplication algorithm, the exponentiation algorithm, and the integer square root algorithm to our general algorithm. For each algorithm, the inputs, the outputs, the number of iterations (n), and the three functions pre(), main() (at iteration i) and post() are identified. Note that for the integer square root algorithm to conform to our general algorithm, its input bits have to come from two A-type inputs. In fact, the bit-by-bit processing of the A-type inputs does not limit the generality of our approach as k n-bit physical A-type inputs can be zipped together to form n k-bit logical inputs if required. In Section VI, we choose these three reference designs to evaluate the accuracy of our model.

C. Design

We focus on the main loop of Algorithm 2 and map it to the general hardware design described in Figs. 3 and 4.

The n iterations are divided between p blocks organized in a pipeline fashion as shown in Fig. 3. Pipeline block number i performs iter(i) successive iterations of the loop where for

TABLE I
SPECIALIZATION OF THE GENERAL ALGORITHM FOR THREE APPLICATIONS

	CSA Montgomery	Exponentiation	Integer Square Root
Inputs	$A_0 = A = (a_{s-1}...a_0)_2$ $B_0 = B = (b_{s-1}...b_0)_2$ $B_1 = M = (m_{s-1}...m_0)_2$	$A_0 = E = (e_{s-1}...e_0)_2$, $B_0 = X$, $B_1 = M$	$A = (a_{s-1}...a_1a_0)_2$ input integer $A_0 = (a_1a_3...a_{s-1})_2$ $A_1 = (a_0a_2...a_{s-2})_2$
Outputs	$R_0 = S = A.B.2^{-n} \bmod M$, $R_1 = C$, $R_2 = D$	$R_0 = Z = X^E \bmod M$, $R_1 = P$	$R_0 = \text{Rem}$, $R_1 = \text{Root}$
n	s	s	$s/2$
pre()	$S = 0$, $C = 0$, $D = B + M$	$Z = 1$, $P = X$	Rem = 0, Root = 0
main()	$I = \text{select}(s_0, c_0, a_i, B, M, D)$ $S, C = S + C + I$ $S = S$ div 2 $C = C$ div 2	**if** $e_i = 1$ **then** $Z = Z.P \bmod M$ $P = P^2 \bmod M$	Root = (Root << 1) $a = (a_{(0,i)}a_{(1,i)})_2$ Rem = (Rem << 2) + a Div = (Root << 1) + 1 **if** Div ≤ Rem **then** Rem = Rem − Div Root = Root + 1
post()	$S = S + C$ **if** $S \geq M$ **then** $S = S - M$	-	-

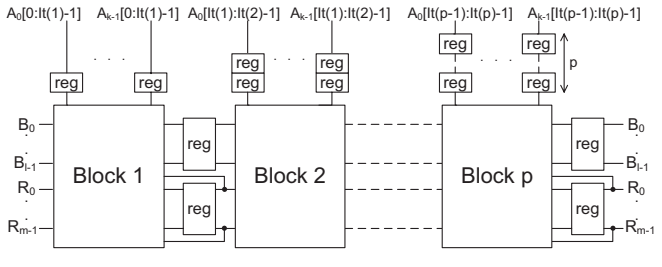

Fig. 3. Pipeline of the parametric hardware description.

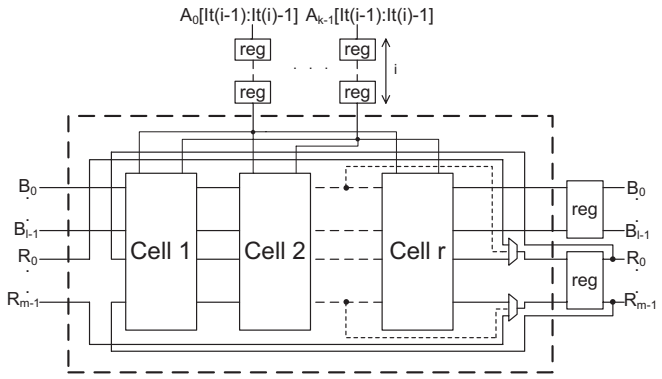

Fig. 4. Inside pipeline block i.

all $i \in [1, p]$

$$\text{iter}(i) = \left\lfloor \frac{n}{p} \right\rfloor + \text{extra}(i) \tag{1}$$

$$\text{extra}(i) = \begin{cases} 1, & \text{if } n \bmod p > i \\ 0, & \text{if } n \bmod p \leq i. \end{cases} \tag{2}$$

The number of iterations is evenly distributed between pipeline blocks. If p does not divide n, the first $n \bmod p$ blocks perform an extra iteration. So block number i processes the bits $It(i-1)$ to $(It(i)-1)$ of inputs A_0 to A_{k-1} where

$$It(0) = 0$$
$$It(i) = \sum_{k=1}^{i} \text{iter}(k) \quad \forall i \in [1, p]. \tag{3}$$

The other inputs to block number i are:

1) the current values of $R_0, \ldots, R_{m-1}$, calculated by block number $(i-1)$ if $i \geq 2$ or given by the pre-computation stage if $i = 1$;
2) the values of $B_0, \ldots, B_{l-1}$.

Each pipeline block consists of a processing cell performing the actual computation, some registers, and extra logic needed for the control. As well as organizing the basic blocks in a pipeline fashion, we allow the processing cell of each block to be replicated in series without buffering as shown in Fig. 4. This feature is relevant if the processing cell consists of only combinational logic. If the operations performed by a cell take several clock cycles, replication will not reduce the number of clock cycles spent in each pipeline block. For instance, given that the operations of a cell take ten clock cycles and we need to perform two iterations in each pipeline block. With one cell, the unique cell performs both iterations one after the other for a total of 20 clock cycles. If we replicate the cell once, the first cell performs the first iteration. The second cell waits for the first to finish then it performs the second iteration. Hence, the total time through the block is still 20 clock cycles.

If each pipeline block has r main processing elements, r iterations are processed in series during one clock cycle. Hence, the number of clock cycles spent in pipeline block i per operation is

$$\text{lat}(i) = \left\lceil \frac{\text{iter}(i)}{r} \right\rceil. \tag{4}$$

We call this number the latency of pipeline block $i \cdot \text{lat}(i)$ is expressed in clock cycles and is therefore independent of the frequency of the design. This equation shows that an extra clock cycle is needed if r does not divide iter(i). In this case,

the results of block i need to be extracted from cell number iter(i) mod r during this extra clock cycle.

Between each pair of blocks, register banks save the different values of $R_0, \ldots, R_{m-1}$ and $B_0, \ldots, B_{l-1}$ to be given as inputs to the next block. The register bank for $R_0, \ldots, R_{m-1}$ is updated at each clock cycle with the results of cell number r or cell number iter(i) mod r. The register bank for $B_0, \ldots, B_{l-1}$ is only updated each time the operation of the preceding block is done. Finally, a triangle of registers at the top ensures that the pipeline blocks are always given the correct values for their corresponding A inputs. In the triangle of registers, the bottom registers connecting to each pipeline block are shift registers. They provide the cells in the pipeline block with the inputs corresponding to the current iteration.

Mapping our general algorithm to this particular hardware structure has three main advantages. First, the parametric nature of this structure (through the two parameters r and p) allows the designer to explore a large design space and therefore to consider different speed/area tradeoffs. Second, the throughput of the design can be improved by increasing the values of r and p. The only limitation is the area available. This is particularly interesting for applications targeting high-speed performance. Finally, our structure is regular and the generation of the pipeline and replication logic is automated.

IV. Modeling of the Design

We develop a model of our parametric design presented in the previous section enabling rapid optimization for throughput under area and speed constraints. This model is a refinement of our previous work presented in [18]. The use of such a model reduces development time as fewer synthesis operations are needed to obtain satisfactory values for r and p.

A. Latency and Throughput

Our module needs to perform n iterations to complete the main computation if no replication is introduced. Let $t_{p,e}$ be the time to perform such an iteration, corresponding to c clock cycles at a frequency f

$$t_{p,e} = \frac{c}{f}. \tag{5}$$

If $c = 1$, the main processing element of a block can be replicated. If $c > 1$, as we cannot replicate the main processing element it takes $n.c$ clock cycles to perform the n iterations. Therefore, the total time to perform the main operation is

$$t_p = \begin{cases} \sum_{i=1}^{p} \text{lat}(i) \cdot t_{p,e}, & \text{if } c = 1 \\ n \cdot t_{p,e}, & \text{if } c > 1. \end{cases} \tag{6}$$

This corresponds to the time taken by the system to compute the first result, that is, the latency of the hardware module in seconds. In practice, replication increases the delay through a pipeline block and therefore reduces the maximum frequency f of the clock as cells are put in series without buffering. This is considered in Section IV-B.

To simplify the equations, we assume that the latency of the pre-computation and post-computation operations is shorter than the latency of a pipeline block. If not the case, the throughput bottleneck is in the slower of the two stages, not in the main computation. The throughput ϕ of the design (number of operations per unit of time when the pipeline is full) depends on both the pipeline length p and the number of replications r. It is inversely proportional to the maximum number of clock cycles spent in a pipeline block

$$\phi = \begin{cases} \dfrac{1}{\left\lceil \frac{\left\lceil \frac{n}{p} \right\rceil}{r} \right\rceil t_{p,e}} = \dfrac{f}{\left\lceil \frac{\left\lceil \frac{n}{p} \right\rceil}{r} \right\rceil}, & \text{if } c = 1 \\ \dfrac{1}{\left\lceil \frac{n}{p} \right\rceil t_{p,e}} = \dfrac{f}{\left\lceil \frac{n}{p} \right\rceil c}, & \text{if } c > 1. \end{cases} \tag{7}$$

Replicating and pipelining a design reduce the effective number of iterations needed to be computed in each block, leading to the following constraint:

$$p \cdot r \leq n. \tag{8}$$

B. Frequency

To simplify, we assume that the pre-processing and post-processing modules are not a frequency bottleneck. Hence, the critical path is in a pipeline block and the minimum period is equal to the delay through a block. In our parametric description, the size of the main processing element in each block is about the same for all p. Even if the size of the control hardware managing the iterations in a block decreases with p, the delay through a block is not likely to decrease significantly. Therefore, we assume that the minimum period of the design does not depend on p. On the contrary, replication has a negative effect on the critical path as the replicated cells are connected together in series. The minimum period can, therefore, be approximated by

$$T_p(r) \approx d_l + r d_c \tag{9}$$

where d_c is the delay through a cell, r is the number of cells in a block, and d_l is the sum of the delays through the logic located before and after the cells (mostly multiplexers). Hence, the frequency of the design is

$$f(r) = \frac{1}{T_p(r)} = \frac{1}{d_l + r d_c}. \tag{10}$$

Let us define

$$f_0 = 1/T_p(1) \tag{11}$$

as the frequency of the design for $r = 1$ and

$$\lambda = d_c/T_p(1) \tag{12}$$

as the ratio of the delay through the basic cell to the minimum period for $r = 1$. We can rewrite the frequency as

$$f(r) = \frac{1/T_p(1)}{1 + d_c/T_p(1)(r-1)} = \frac{f_0}{1 + \lambda(r-1)}. \tag{13}$$

C. Area

We decompose the area taken by our design into two parts.

1) A_l the area taken by logic.
2) A_r the area taken by registers.

Let us first consider the area taken by the logic A_l. Let A_{cell} be the logic area of a cell for $r = 1$, $A_{\text{mux},2}$ and $A_{\text{mux},3}$ be the area taken, respectively, by a 2-to-1 and a 3-to-1 bit multiplexer. We recall that ρ_i is the bitwidth of the output R_i and m is the number of such outputs. As shown in Fig. 4, the multiplexers required at the inputs of the R register bank are either two-input or three-input multiplexers, depending on the need for an extra clock cycle in pipeline block i. Hence, the logic area of pipeline block i is

$$A_{\text{block}}(i) = r.A_{\text{cell}} + \sum_{k=0}^{m-1} \rho_k.A_{\text{mux},2+\text{extra}}(i). \tag{14}$$

Each register of the top register triangle needs either a two-input or a three-input multiplexer depending on whether they are shift registers. The part of the register triangle supplying inputs to pipeline block number i is of size iter(i). We recall that k is the number of A inputs. Hence, the logic area of the register triangle is

$$A_{\text{triangle}} = k \sum_{i=1}^{p} \text{iter}(i).\left((i-1).A_{\text{mux},2} + A_{\text{mux},3}\right). \tag{15}$$

Let A_{pre} and A_{post} be the fixed logic area taken, respectively, by the pre-computation and post-computation modules. We can deduce the total logic area

$$A_l = A_{\text{pre}} + A_{\text{triangle}} + \sum_{i=1}^{p} A_{\text{block}}(i) + A_{\text{post}}. \tag{16}$$

To calculate A_r, we need to derive the total register size S. The total size of the register banks between two pipeline blocks is

$$S_d = \sum_{i=0}^{l-1} \beta_i + \sum_{i=0}^{m-1} \rho_i. \tag{17}$$

The size of a register in the top triangle of registers is for pipeline block i

$$S_p(i) = k.\text{iter}(i). \tag{18}$$

Let S_{cell} be the total size of the registers internal to a cell, S_{pre} and S_{post} be, respectively, the total size of the registers of the pre-computation and post-computation modules. The total register size is

$$S = S_{\text{pre}} + p.(S_d + r.S_{\text{cell}}) + \sum_{i=1}^{p} i.S_p(i) + S_{\text{post}}. \tag{19}$$

Finally, we have

$$A_r = S.A_{r,e} \tag{20}$$

where $A_{r,e}$ is the area taken by a one-bit register.

TABLE II
DETERMINATION OF THE DESIGN PARAMETERS

Parameters	Determination
A inputs number (k), bitwidth (n)	Application-dependent
B inputs number (l), bitwidths ($\beta_0, \ldots, \beta_{l-1}$)	Application-dependent
R outputs number (m), bitwidths ($\rho_0, \ldots, \rho_{m-1}$)	Application-dependent
Iteration clock cycles (c)	Application-dependent
Minimum frequency $f_{\min}$	Design choice
Maximum logic area ($A_{l,\max}$)	Device-dependent
Maximum register area ($A_{r,\max}$)	Device-dependent
Basic cell area (A_{cell})	Pre-synthesis
Basic cell registers (S_{cell})	Pre-synthesis
Pre/post-computation area (A_{pre}, A_{post})	Pre-synthesis
Pre/post-computation registers (S_{pre}, S_{post})	Pre-synthesis
Area of a 1-bit register ($A_{r,e}$)	Pre-synthesis
Multiplexers area ($A_{\text{mux},2}$ and $A_{\text{mux},3}$)	Pre-syntheses
Frequency parameters (f_0, λ)	Interpolation or direct

D. Constraints

The following constraints are used:

1) maximum area available for logic $A_{l,\max}$;
2) maximum area available for registers $A_{r,\max}$;
3) minimum frequency $f_{\min}$ at which we want the design to run.

E. Throughput Optimization

The problem of optimizing the throughput of the design can be formulated as follows:

$$\max \quad \phi = \begin{cases} \dfrac{1}{\left\lceil \frac{\left\lceil \frac{n}{p} \right\rceil}{r} \right\rceil t_{p,e}} = \dfrac{f}{\left\lceil \frac{\left\lceil \frac{n}{p} \right\rceil}{r} \right\rceil}, & \text{if } c = 1 \\ \dfrac{1}{\left\lceil \frac{n}{p} \right\rceil t_{p,e}} = \dfrac{f}{\left\lceil \frac{n}{p} \right\rceil c}, & \text{if } c > 1 \end{cases}$$

$$\text{s.t.} \quad A_l \le A_{l,\max} \qquad A_r \le A_{r,\max} \qquad f \ge f_{\min} \qquad p.r \le n. \tag{21}$$

Given a search interval for r and p, this problem is solved easily by exhaustive search.

V. OPTIMIZING THE DESIGN

We show how the parameters of our model can be determined, and an optimization process based on our model.

A. Determining the Values of the Parameters

Our model contains many parameters which affect the values of r and p maximizing the throughput. Table II summarizes how each parameter is determined. The number of A inputs (k), the number of B inputs (l), the number of

outputs (m), and their respective bitwidths (n, $\beta_0, \ldots, \beta_{l-1}$, $\rho_0, \ldots, \rho_{m-1}$), and the number of clock cycles c needed to complete an iteration, depend on the application. Hence, we can obtain them easily once the design of the hardware module is fixed. The minimum frequency $f_{\min}$ at which our module runs is a design choice. The maximum area available for registers $A_{r,\max}$ and for logic $A_{l,\max}$ are device-dependent. For an FPGA, $A_{r,\max}$ is the maximum number of registers and $A_{l,\max}$ is the maximum number of look-up tables (LUTs) available. For an ASIC, $A_{r,\max}$ and $A_{l,\max}$ can be grouped together to provide the maximum area available given in μm^2, in number of cells or in the technology-independent notion of a register bit equivalent [19].

The area of a basic cell A_{cell}, of the pre-computation and post-computation modules (A_{pre} and A_{post}) can be approximated by using *a priori* knowledge of the design or found by running a single synthesis operation for $r = 1$ and $p = 1$. This is also the case for the registers internal to a cell S_{cell} and the registers of the pre-computation and post-computation modules (S_{pre} and S_{post}). The area of a one-bit register ($A_{r,e}$), 2-to-1 and 3-to-1 multiplexers ($A_{\text{mux},2}$ and $A_{\text{mux},3}$) are easily determined by running short synthesis operations.

The frequency parameters f_0 and λ can be obtained by two different means. The first solution is to run some synthesis operations in order to get the values of f for a small set of chosen r and find f_0 and λ by interpolation. The second solution is to run a single synthesis operation to find the value of f_0 and compute λ directly using 12, provided that we can estimate the delays through a pipeline block and a replicated cell. In our current implementation of the optimization process, the different parameters need to be extracted by the designer.

If the search intervals for r and p are large enough, the time taken by the extra synthesis operations (we call them pre-synthesis operations) is negligible compared to the time it would take to synthesize the design for every possible value of the tuple (r, p) in the search intervals. More precisely, running three or four pre-synthesis operations for uniformly distributed values of r in the search interval is usually enough to get a good approximation of λ by interpolation. The pre-synthesis time needed to determine the multiplexers area is negligible. Moreover, one of the pre-synthesis operations used to determine the frequency parameters also gives us the basic cell area A_{cell}. Hence for $r \cdot p > 4$, using our model should be faster than a complete search through the design space. Results demonstrating this claim are shown in Section VI.

B. Design Generation and Optimization Process

Given the different application-dependent parameters of the design and the number of pipeline stages p and replications r, a script automatically generates RTL code for our general pipelined and replicated hardware mapping as well as the corresponding control logic, which greatly simplifies the work of the designer. Only the basic cell, the pre-computation and post-computation logic are defined by the designer for each application. In the examples presented in Section VI, these modules are coded in Verilog. A higher level hardware language, such as Handel-C or Xilinx HLS could be used to reduce the design time further.

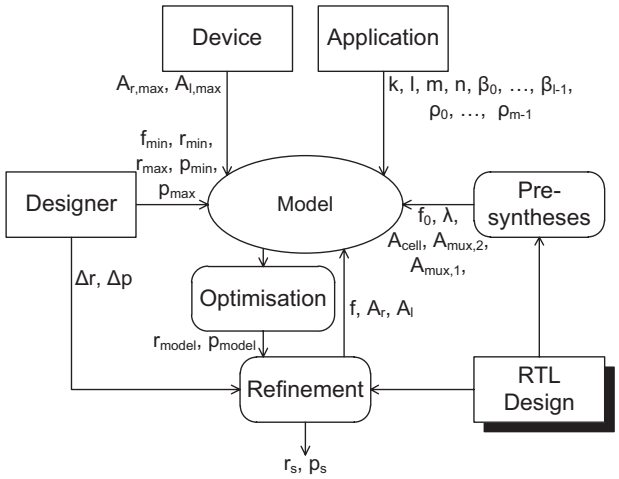

Fig. 5. Optimization process.

The complete optimization process of our parametric design is depicted in Fig. 5. The aim of this process is to find satisfactory values for the parameters r and p, that is values that are close to the real optima, in a short amount of time. First, the different parameters of the model are determined using the methods described in Table II. Given a search interval for r and p, respectively, $[r_{\min}, r_{\max}]$ and $[p_{\min}, p_{\max}]$, the throughput optimization of 21 gives us the optimal values of r and p according to the model. We call them r_{model} and p_{model}. The values r_{model} and p_{model} are then given to the refinement process. This optional process performs a small number of synthesis operations around r_{model} and p_{model} to refine the solution by giving the real post-synthesis values of f, A_r and A_p to the model. The refinement intervals for r and p are, respectively, $[r_{\text{model}} - \Delta r, r_{\text{model}} + \Delta r]$ and $[p_{\text{model}} - \Delta p, p_{\text{model}} + \Delta p]$. The outputs of this process are r_s and p_s. The pre-synthesis, optimization, and refinement processes are implemented as a Python software tool, which interfaces with Xilinx synthesis tools.

VI. Results

In this section, we evaluate the accuracy and the benefits of our model against a complete search through the design space.

A. Accuracy of the Model

We consider three applications presented in Table I.

1) 512-bit CSA-based Montgomery multiplication.
2) 128-bit modular exponentiation.
3) 512-bit integer square root.

The hardware mappings of these designs are synthesized for a Xilinx Spartan 6 XC6SLX45T-FGG484-3 with XST 13.2 optimizing for area, without resource sharing and equivalent register removal. All the synthesis operations are run on an Intel Core i7 950 CPU at 3.07-GHz machine with 6 GB of DDR3 memory.

The basic cells of the Montgomery multiplier and the integer square root modules only consist of combinational

TABLE III
ACCURACY OF OUR MODEL

	LUTs Error (%)			Registers Error (%)			Frequency Error (%)			Throughput Error (%)			Corrected Throughput Error (%)		
	Max	Mean	Std	Max	Mean	Std	Max	Mean	Std	Max	Mean	Std	Max	Mean	Std
Montgomery	1.18	0.32	0.20	0.42	0.26	0.09	21.03	1.87	2.87	128.85	17.38	26.03	21.03	2.33	3.43
Exponentiation	4.43	3.32	0.68	1.23	1.19	0.02	15.73	14.75	2.96	15.73	14.75	2.96	15.73	14.75	2.96
Square root	3.41	0.60	0.66	1.24	0.97	0.12	0.68	0.34	0.14	0.68	0.34	0.14	0.68	0.34	0.14

logic ($c = 1$). Hence, replication can be introduced. We synthesize these two designs for all the 256 combinations of the number of replications (r) and pipeline stages (p) in the respective intervals $[1, 16]$ and $[1, 16]$. The basic cell of the modular exponentiation module, which is basically a modular multiplier, is sequential ($c > 1$). Therefore, replicating this cell is irrelevant. We synthesize this design for $r = 1$ and p in $[1, 32]$. For the multiplier and the square root modules, we determine the frequency parameters by interpolation using three points: $(p, r) = (1, 1)$, $(p, r) = (1, 8)$, and $(p, r) = (1, 16)$. The frequency of the exponentiation module is fixed to its value for $p = 1$.

For our three applications, Table III reports the maximum, average, and standard deviation of the relative prediction errors[1] of our model across all the combinations of the parameters p and r for the number of LUTs, the number of registers, the maximum frequency, and the throughput. The maximum errors for the number of LUTs and registers are very low over the three applications (1.18%–4.43%, and 0.42%–1.24%, respectively). The maximum frequency error is more substantial for the Montgomery multiplication and the exponentiation module. We suspect that this error is due to uncontrollable optimizations performed by the synthesis tool on routing delays that sometimes make the frequency depend on the number of pipeline stages p. This dependency on p does not follow a general trend and therefore cannot be modeled effectively. As shown in Fig. 6, for the Montgomery multiplier, high frequency errors are localized to a few (r, p) tuples.[2] This is not the case for the exponentiation module whose average frequency error is high and close to the maximum frequency error. Here, the reference point chosen to determine f_0 ($(r, p) = (1, 1)$) has a maximum frequency, which is around 15% lower than for the other values of p, leading to a constant error. The error could be improved by picking a different reference point $(r, p) = (1, 2)$ for example.

The throughput error of the Montgomery multiplier is very high. Our model supposes that the latency of the pre-computation stage is shorter than the latency of any pipeline block. However, in this application, the pre-computation stage performs a pipelined addition, which takes eight clock cycles. For r and p such that $\lceil \lceil n/p \rceil / r \rceil < 8$, the latencies of the pipeline blocks are less than eight and the throughput of the module is fixed and only determined by the latency of the pre-computation stage. This invalidates 7. These values of

[1]Computed as $|v - v_{\text{model}}|/v$ where v is the value considered.
[2]Here, the errors are computed as $(f_{\text{model}} - f)/f$ to show their directions.

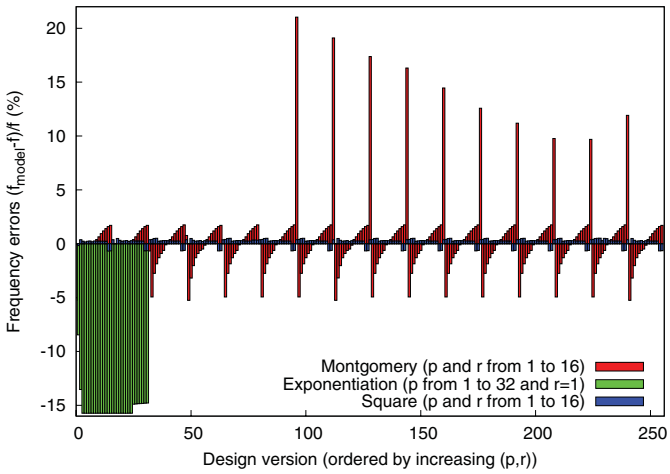

Fig. 6. Distribution of the frequency errors for the different designs.

r and p should be avoided as they lead to pointless area-consuming designs. The last column of Table III shows the corrected throughput errors, that is, when not taking into account these designs. After such corrections, the throughput errors are similar to the frequency errors.

B. Design Space Exploration

We evaluate the prediction capabilities of our model. The Spartan 6 used has $A_{l,s6} = 27\,288$ LUTs and $A_{r,s6} = 54\,576$ flip-flops (FFs). For each of our three applications, we use our model to find the design with maximum throughput for all the combinations of $f_{\min}$ (in MHz) in $[0, 25, 50, 100]$ and $(A_{l,\max}, A_{r,\max})$ (respectively, in LUTs and FFs) in $[(A_{l,s6}/4, A_{r,s6}/4), (A_{l,s6}/2, A_{r,s6}/2), (A_{l,s6}, A_{r,s6}), (\infty, \infty)]$. Hence, for each application, we perform 16 different optimizations. Note that setting $f_{\min} = 0$ MHz corresponds to not putting any constraint on the frequency. Setting $(A_{l,\max}, A_{r,\max}) = (\infty, \infty)$ corresponds to assuming that the FPGA has unlimited area.

Table IV presents our results. The design space size corresponds to the number of synthesis operations needed for a full search through the design space without using our method. The total synthesis time is the time taken to perform this search. The two other synthesis times reported are the average total synthesis time over the 16 optimizations performed using our model, respectively, before and after refinement. Without refinement, our method can speed up the design space exploration by an average of 419 times for the CSA Montgomery application and 200 times for the integer square

TABLE IV
EVALUATION OF PREDICTION CAPABILITIES OF THE MODEL

	Design Space Size	Total Synthesis Time (min)	Before Refinement			After Refinement			
			Max Throughput Error (%)	Average Synthesis Time (min)	Average Synthesis Speedup (x)	Δr	Δp	Average Synthesis Time (min)	Average Synthesis Speedup (x)
Montgomery	256	2489	25.48	7	419	2	2	33	96
Exponentiation	32	49	37.88	2	34	0	1	5	13
Square root	256	698	9.09	4	200	0	1	8	91

root. The lower average synthesis speedup of 34 times for the exponentiation application is directly linked to the smaller size of the design space explored.

Without refinement, the optimum design is not always found by our model. For the Montgomery multiplier module, our worst estimation is two pipeline stages and two replications off the optimum design. For the exponentiation and the integer square root modules, it is only one pipeline stage off. This leads to a throughput error (error between the optimum throughput and the throughput of the design given by the model) between 9% and 25%. In order to find the optimum design, extra syntheses need to be performed. This reduces the average synthesis speedups by a factor of 2 to 4. However, even with this reduction in speedup, our method can still save hours of synthesis time. In particular, the design space exploration time for the CSA Montgomery application is reduced from almost 2 days to only half an hour.

Note that our results assume serial execution of the syntheses on the same machine. Synthesis jobs can also be run on different machines in parallel. However, a parallel version of our method should also be faster than a parallel search through the design space, since the synthesis time of each pre-synthesis operation is shorter than the synthesis time of most other points in the design space.

VII. CONCLUSION

In this paper, we showed how we automatically map a general type of algorithms to a parametric hardware design using pipelining and replication. A technology-independent model of our general design was developed. It allows rapid hardware mapping and optimization of the algorithm by running only a few numbers of pre-synthesis operations and therefore reduces time-to-market. We presented an optimization method based on the model and evaluated on three different applications implemented on a Xilinx Spartan 6 XC6SLX45T FPGA: a CSA-based Montgomery multiplier, a modular exponentiation module, and an integer square root module. Our model facilitates design space exploration; it can quickly predict the area taken by our designs with less than 5% of error, and their maximum frequencies and throughputs with less than 22% of error. Our optimization method is up to 96 times faster than a full search through the design space.

Current and future work includes extending our model to optimize for area, area-time, and taking into account power consumption; applying our optimization method to other designs and to other technologies such as ASICs; synthesizing our reference designs optimizing for speed instead of area; and automating our optimization process further so that the general algorithm pattern can be detected and the parameters extracted automatically from a high-level description.

REFERENCES

[1] T. Becker, W. Luk, and P. Y. Cheung, "Parametric design for reconfigurable software-defined radio," in *Proc. 5th Int. Workshop Reconfig. Comput. Archit. Tools Appl.*, Mar. 2009, pp. 15–26.

[2] A. Le Masle, W. Luk, J. Eldredge, and K. Carver, "Parametric encryption hardware design," in *Proc. Reconfig. Comput. Archit. Tools Appl.*, Jul. 2010, pp. 68–79.

[3] S. Yusuf, W. Luk, M. Sloman, N. Dulay, E. Lupu, and G. Brown, "Reconfigurable architecture for network flow analysis," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 1, pp. 57–65, Jan. 2008.

[4] R. Enzler, T. Jeger, D. Cottet, and G. Tröster, "High-level area and performance estimation of hardware building blocks on FPGAs," in *Proc. 10th Int. Workshop Field Program. Logic Appl. Roadmap Reconfig. Comput.*, Aug. 2000, pp. 525–534.

[5] P. Schumacher and P. Jha, "Fast and accurate resource estimation of RTL-based designs targeting FPGAs," in *Proc. Int. Conf. Field Program. Logic Appl.*, Sep. 2008, pp. 59–64.

[6] D. Kulkarni, W. A. Najjar, R. Rinker, and F. J. Kurdahi, "Compile-time area estimation for LUT based FPGAs," *ACM Trans. Design Autom. Electron. Syst.*, vol. 11, no. 1, pp. 104–122, Jan. 2006.

[7] C. Shi, J. Hwang, S. McMillan, A. Root, and V. Singh, "A system level resource estimation tool for FPGAs," in *Proc. Int. Conf. Field Program. Logic Appl.*, Jun. 2004, pp. 424–433.

[8] T. Becker, W. Luk, and P. Y. K. Cheung, "Energy-aware optimisation for run-time reconfiguration," in *Proc. 18th IEEE Annu. Int. Symp. Field Program. Custom Comput. Mach.*, May 2010, pp. 55–62.

[9] B. So, M. W. Hall, and P. C. Diniz, "A compiler approach to fast hardware design space exploration in FPGA based systems," *ACM SIGPLAN Not.*, vol. 37, no. 5, pp. 165–176, Jun. 2002.

[10] M. Weinhardt, "Compilation and pipeline synthesis for reconfigurable architectures," in *Proc. Reconfig. Archit. Workshop*, 1997, pp. 1–8.

[11] K. Bondalapati and V. K. Prasanna, "Mapping loops onto reconfigurable architectures," in *Proc. 8th Int. Workshop Field Program. Logic Appl.*, 1998, pp. 268–277.

[12] H. Styles, D. Thomas, and W. Luk, "Pipelining designs with loop-carried dependencies," in *Proc. IEEE Int. Conf. Field Program. Technol.*, Dec. 2004, pp. 255–262.

[13] N. Weaver, Y. Markovskiy, Y. Patel, and J. Wawrzynek, "Post-placement C-slow retiming for the Xilinx Virtex FPGA," in *Proc. 11th ACM Int. Symp. Field Program Gate Arrays*, Feb. 2003, pp. 185–194.

[14] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 170, pp. 519–521, Apr. 1985.

[15] *Digital Signature Standard*, FIPS PUB Standard 186-3, 2009.

[16] A. Le Masle, W. Luk, and C. A. Moritz, "Parametrized hardware architectures for the Lucas primality test," in *Proc. Int. Conf. Embedded Comput. Syst.*, Jul. 2011, pp. 124–131.

[17] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, 1st ed. New York, USA: Oxford Univ. Press, 2000, pp. 345–352.

[18] A. Le Masle and W. Luk, "Design space exploration of parametric pipelined designs," in *Proc. 21st IEEE Int. Conf. Appl. Specific Syst. Archit. Process.*, Jul. 2010, pp. 47–54.

[19] J. Mulder, N. Quach, and M. Flynn, "An area model for on-chip memories and its application," *IEEE J. Solid-State Circuits*, vol. 26, no. 2, pp. 98–106, Feb. 1991.

Adrien Le Masle received the M.Sc. degree in advanced computing from the Department of Computing, Imperial College London, London, U.K., where he is currently pursuing the Ph.D. degree from the Department of Computing.

His current research interests include reconfigurable computing, high performance computing, design space exploration, cryptography, and side-channel attacks.

Wayne Luk (F'09) received the M.A., M.Sc., and D.Phil. degrees in engineering and computing science from the University of Oxford, Oxford, U.K.

He is a Professor of computer engineering with Imperial College London, London, U.K. He was a Visiting Professor with Stanford University, Stanford, CA, USA. His current research interests include theory and practice of customizing hardware and software for specific application domains, such as multimedia, networking, and finance.